Hardware-Oriented Numerics

High Performance FEM Simulation of PDEs

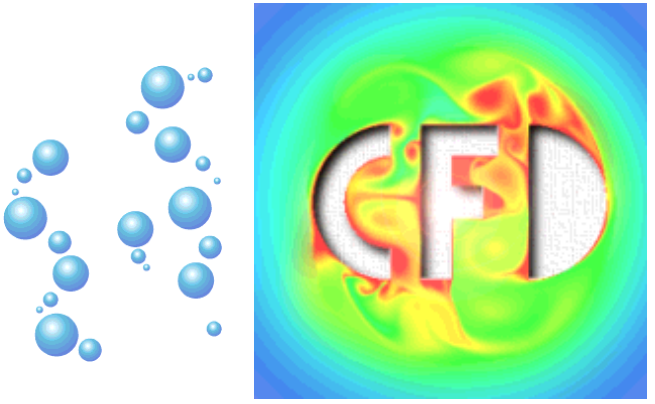
Stefan Turek

Institut für Angewandte Mathematik, Univ. Dortmund

http://www.mathematik.uni-dortmund.de/LS3

http://www.featflow.de

- Performance problems
- Adaptivity/ScaRC solver
- GPU computing



Classification of PDE software:

For education: MATLAB, MAPLE, MATHEMATICA, etc.

⇒ **'play–around'** *with Mathematics, 'easy' user interface*
⇒ *simple, but robust algorithms, easy applicable*
⇒ *'independent of implementation/language'*

For research: DIFFPACK, UG, DEAL, FEMLAB, FEAT, etc.

⇒ *open for numerical and algorithmic changes*
⇒ **flexible**, *but robust data structures,* **reusable** *components*
⇒ *'independent of user interface'*

For "real life" applications: ???

⇒ *'optimal play–together of numerics* **and** *implementation'*
⇒ *PC/Workstation/LINUX-Cluster/Supercomputer*
⇒ *robustness and* **efficiency**

What is:

Hardware-Oriented Numerics for PDE's ?

Answer (I)

It is more than ”better Numerics” and ”better Implementation” on High Performance Computers !

Critical quantity: ‘Total Efficiency !’

Answer (II)

What is the "Total Efficiency" ?

'High (guaranteed) accuracy for user-specific quantities with minimal #d.o.f.s ($\sim N$) via fast and robust solvers - for a wide class of parameter variations - with 'optimal' ($\sim O(N)$) numerical complexity while exploiting a significant percentage of the available huge sequential/parallel GFLOP/s rates at the same time'

How to measure "Total Efficiency" ?

Corresponding Key Technologies

'A posteriori error control/adaptive meshing'

'Iterative (parallel) solution strategies'

'Operator-splitting for coupled problems'

Recent Trends in Processor Technology:

'Enormous improvements in Processing Data'

'Much lower advances in Moving Data'

1 PC in 10 years $\approx$ 1 JUMP/IBM today !!!

Parallel PFLOP/s computers in 5 years !!!

Questions:

'Can we use this enormous computing power ?'

OK for LINPACK, etc.

'How to achieve a high "Total Efficiency" for Numerics for PDE's ?'

For iterative solvers, etc.???

Sequential Performance (0)

Linpack Test (MFlop/s for TOP500 list)

AMD Athlon XP 1500+ with 1,333 MHz core frequency

- Case 1: self compiled BLAS/LAPACK libraries (F77)
- Case 2: vendor delivered BLAS/LAPACK, partially assembler (SUSE)
- Case 3: tuned BLAS/LAPACK with CPU extensions like SSE (ATLAS)

$\#unknowns$	case 1	case 2	case 3
500	323	477	940
1,000	275	481	1,119
3,000	116	152	1,362

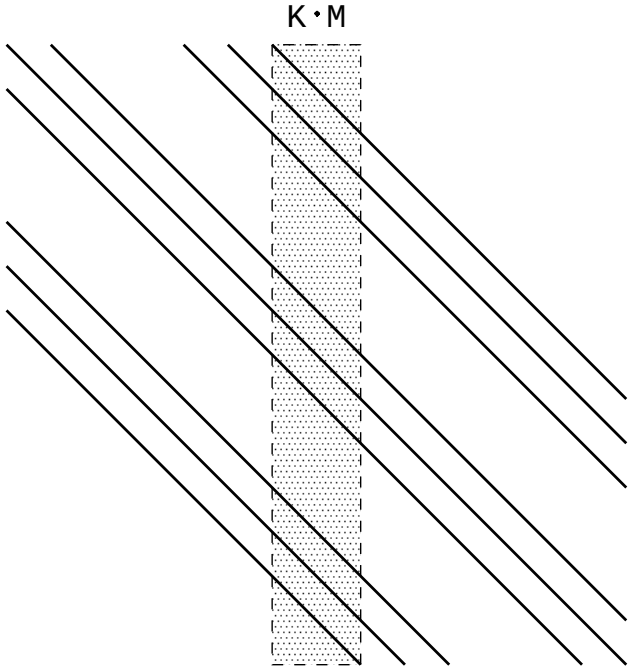
OK !

Main component: 'Sparse' MV Application

SPARSE **Matrix-Vector techniques** ('indexed DAXPY')

```
    DO 10 IROW=1,N
    DO 10 ICOL=KLD(IROW),KLD(IROW+1)-1
10 Y(IROW)=DA(ICOL)*X(KCOL(ICOL))+Y(IROW)
```

SPARSE BANDED **Matrix-Vector techniques**



Sequential Performance (I)

Sparse MV multiplication in (sequential) **FEATFLOW**:

Computer	#Unknowns	CM	TL	STO	ILU-CM	ILU-TL	ILU-STO
	8,320	147	136	**116**	90	76	72
DEC 21264	33,280	125	105	**100**	86	73	63
(667 MHz)	133,120	81	71	**58**	81	52	55
'EV67'	532,480	60	51	**21**	40	35	22
	2,129,920	58	47	**13**	38	30	14
	8,519,680	58	45	**10**	36	30	11

Sequential Performance (II)

'Generalized Tensorproduct' meshes

2D case	NEQ	STO (ROW)	SBB-V	SBB-C	MGTRI-V	MGTRI-C
Sun V40z	65^2	1521 (422)	1111	1605	943	1178
(1800 MHz)	257^2	264 (106)	380	1214	446	769
'**Opteron**'	1025^2	**197 (54)**	**362**	**1140**	**333**	**570**
IBM POWER4	65^2	1521 (845)	2064	3612	1386	1813
(1700 MHz)	257^2	1100 (227)	1140	3422	1048	1645
'**Power**'	1025^2	**390 (56)**	**550**	**2177**	**622**	**1138**

SPARSE **MV techniques (**STO/ROW**)**

MFLOP/s rates vs. 'Peak Performance', problem size + numbering ???

Local Adaptivity !!!

SPARSE BANDED **MV techniques (**SBB**)**

'Supercomputing' (up to 1 GFLOP/s) vs. FEM for complex domains ???

Sequential Performance (III)

Vectorization (preliminary):

2D	NEQ	STO/TL	SBB-V/C	MGTRI-V/C	TRI-V/C
NEC SX	65^2	423/845	1203/1313	447/372	85/62
(500 MHz)	257^2	440/777	1397/2351	479/462	85/84
'**SX-6i**'	1025^2	442/801	1778/3186	491/527	87/87

Development of 'new' methods

$\Uparrow$

"SPAI" preconditioner ($\sim$ pure MV multiplication)
"Cyclic Reduction" preconditioner

Summary (I)

‘It is non-trivial to reach the Sequential Peak Performance with modern (= high numerical efficiency) PDE tools !!!’

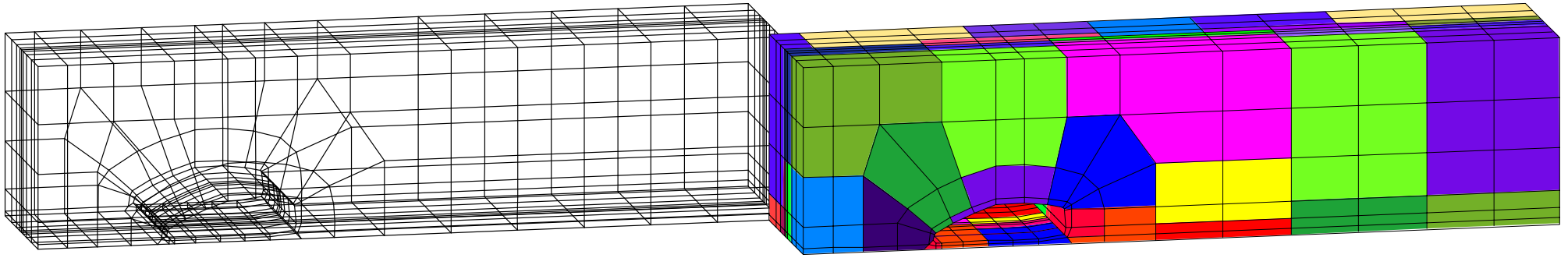
‘Local adaptivity with unstructured meshes ?’

‘Sparse matrix-vector applications ?’

‘Memory-intensive data/matrix/solver structures ?’

Parallel Performance (I)

‘Complex (anisotropic) ASMO3D configuration’



‘(Moderate) mesh anisotropies (AR = 20)’

Parallel Performance (II)

FEATFLOW (F77) vs. parallel version PP3D++

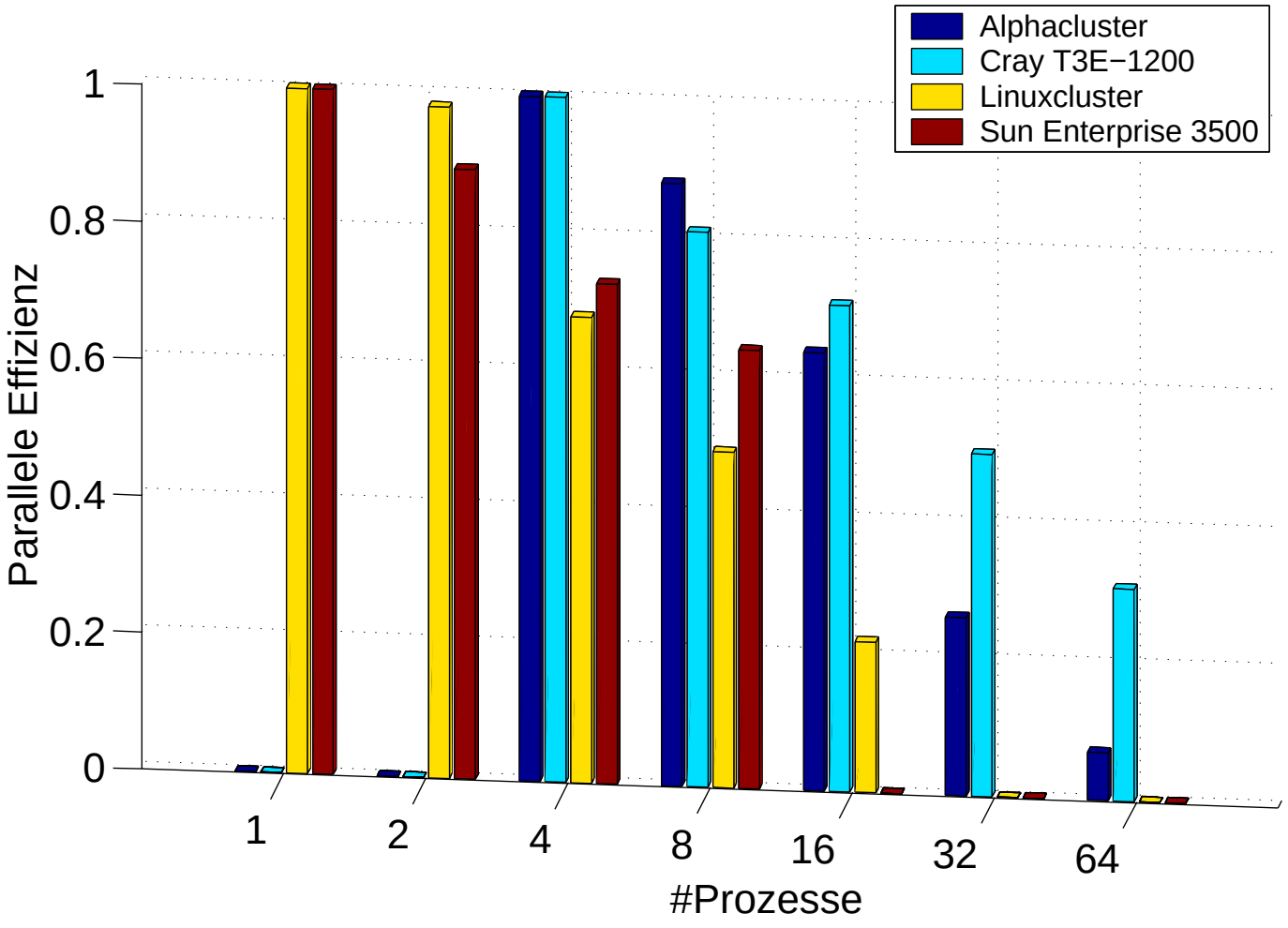
implementation	architecture	d.o.f.		cpu time
		space	time	[min]
sequential (F77)	Alpha ES40	5,375,872	1,455	**2086**
parallel (1 node)	Alpha ES40	5,375,872	1,446	**6921**
parallel (4 nodes)	Alpha ES40	5,375,872	1,466	2151
parallel (4 nodes)	Cray T3E-1200	5,375,872	1,466	4620
parallel (32 nodes)	Cray T3E-1200	5,375,872	1,522	**831**
parallel (70 nodes)	Cray T3E-1200	5,375,872	1,502	615
parallel (130 nodes)	Cray T3E-1200	5,375,872	1,514	688

'Problems due to F77 vs. C++'

'Problems due to communication'

'Problems due to Pressure Poisson multigrid solver'

Parallel Performance (III)



	1 P.	2 P.	4 P.	8 P.	16 P.	32 P.	64 P.
%Comm.	10%	24%	36%	45%	47%	55%	56%
#PPP-IT	2,2	3,0	3,9	4,9	5,2	5,7	6,2

Summary (II)

‘It is non-trivial to reach the Parallel Peak Performance with modern (= high numerical efficiency) PDE tools !!!’

‘Communication behaviour of (LINUX) clusters ?’

‘(Only) Blockwise smoothers in parallel ?’

‘Mesh anisotropies ?’

Summary (III)

‘Special requirements for numerical

and algorithmic approaches in

correspondance to existing hardware !!!’

$\Downarrow$

‘Hardware-Oriented Numerics for PDEs (?)’

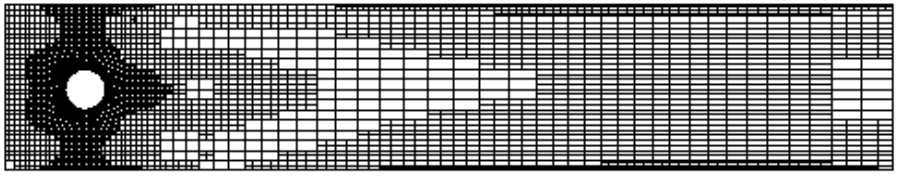
$\Downarrow$

FEAST Project

Special Techniques

I) *Patch-oriented adaptivity*

'Many' tensorproduct grids (SBB)
'Few' unstructured grids (SPARSE)



II) *Generalized MG-DD solver:* SCARC

Exploit locally 'regular' structures **(efficiency)**
Recursive 'clustering' of anisotropies **(robustness)**

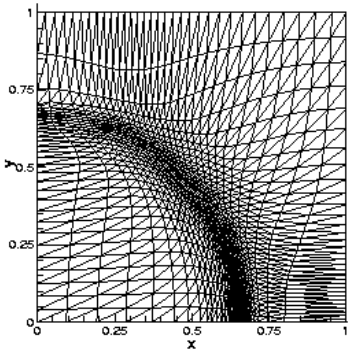
'Strong local solvers improve global convergence !'

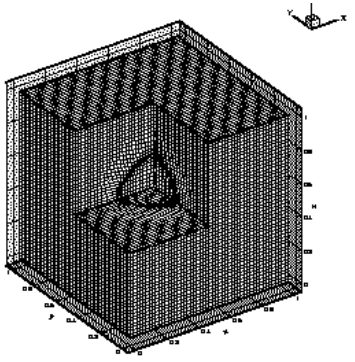
'Exploit locally regular structures !!!'

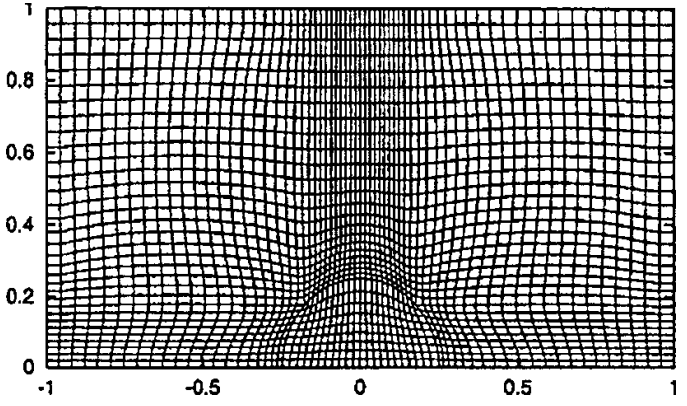
Concepts for Adaptive Meshing

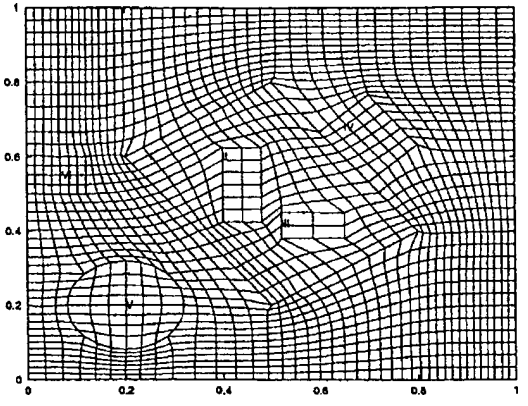
1) macro-oriented adaptivity

2) patchwise ‘deformation’ adaptivity

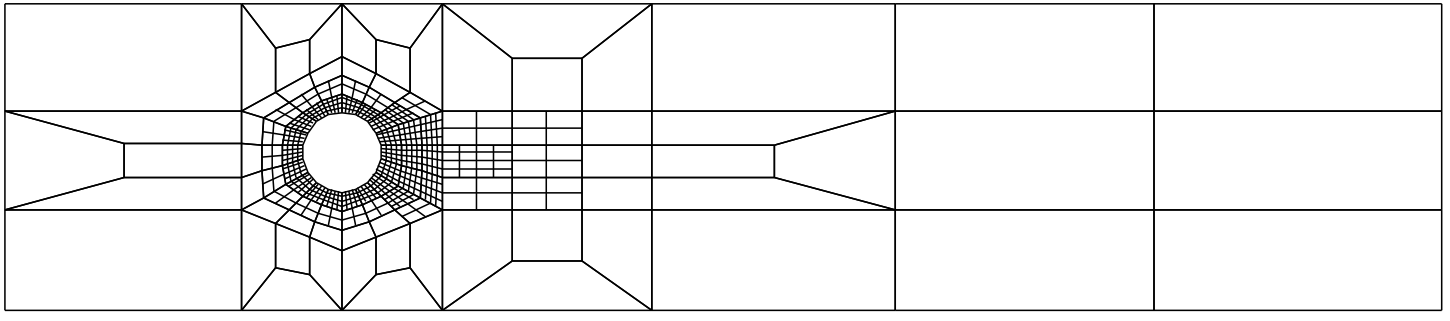




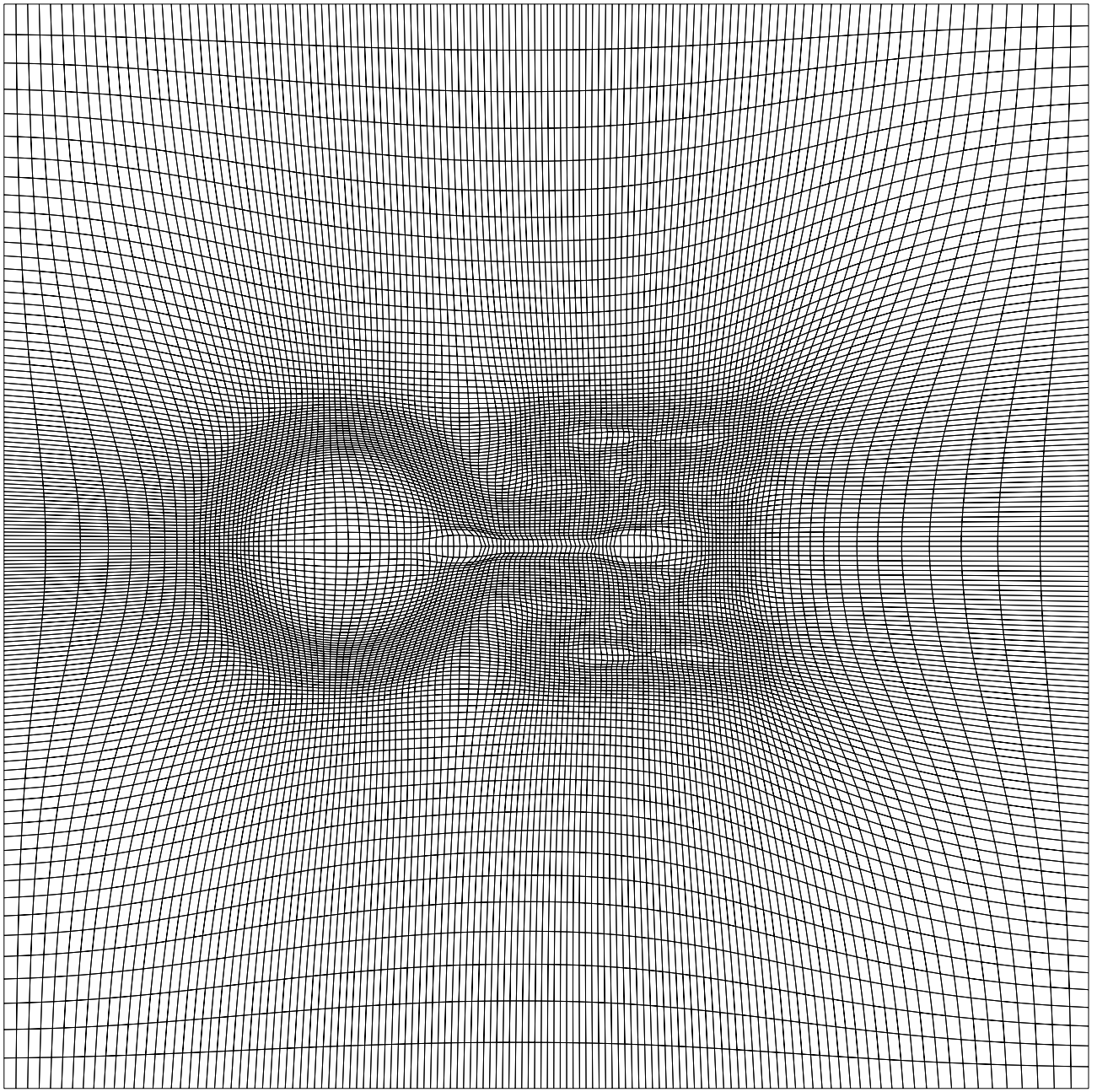




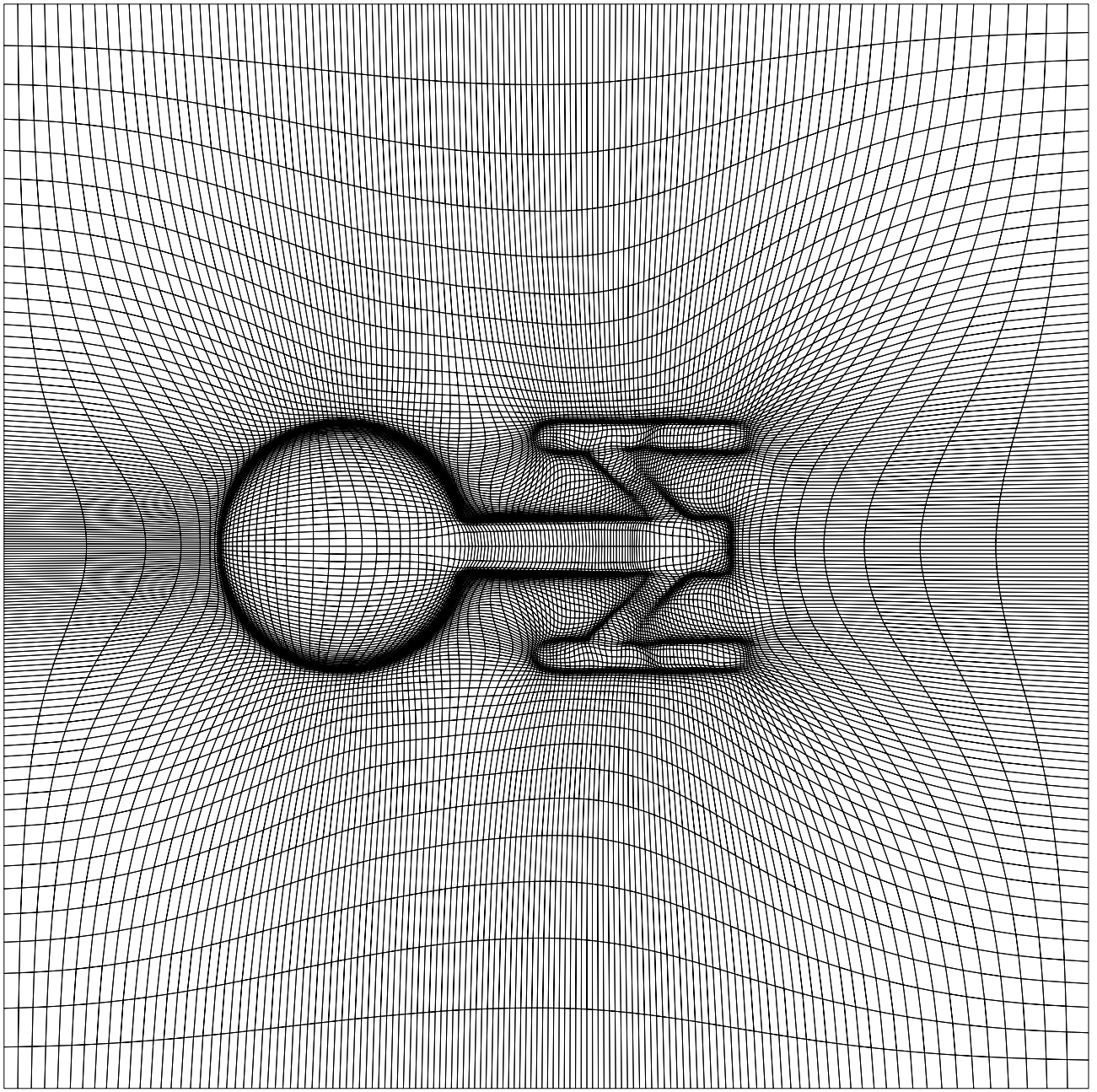
3) (patchwise) ‘local’ adaptivity



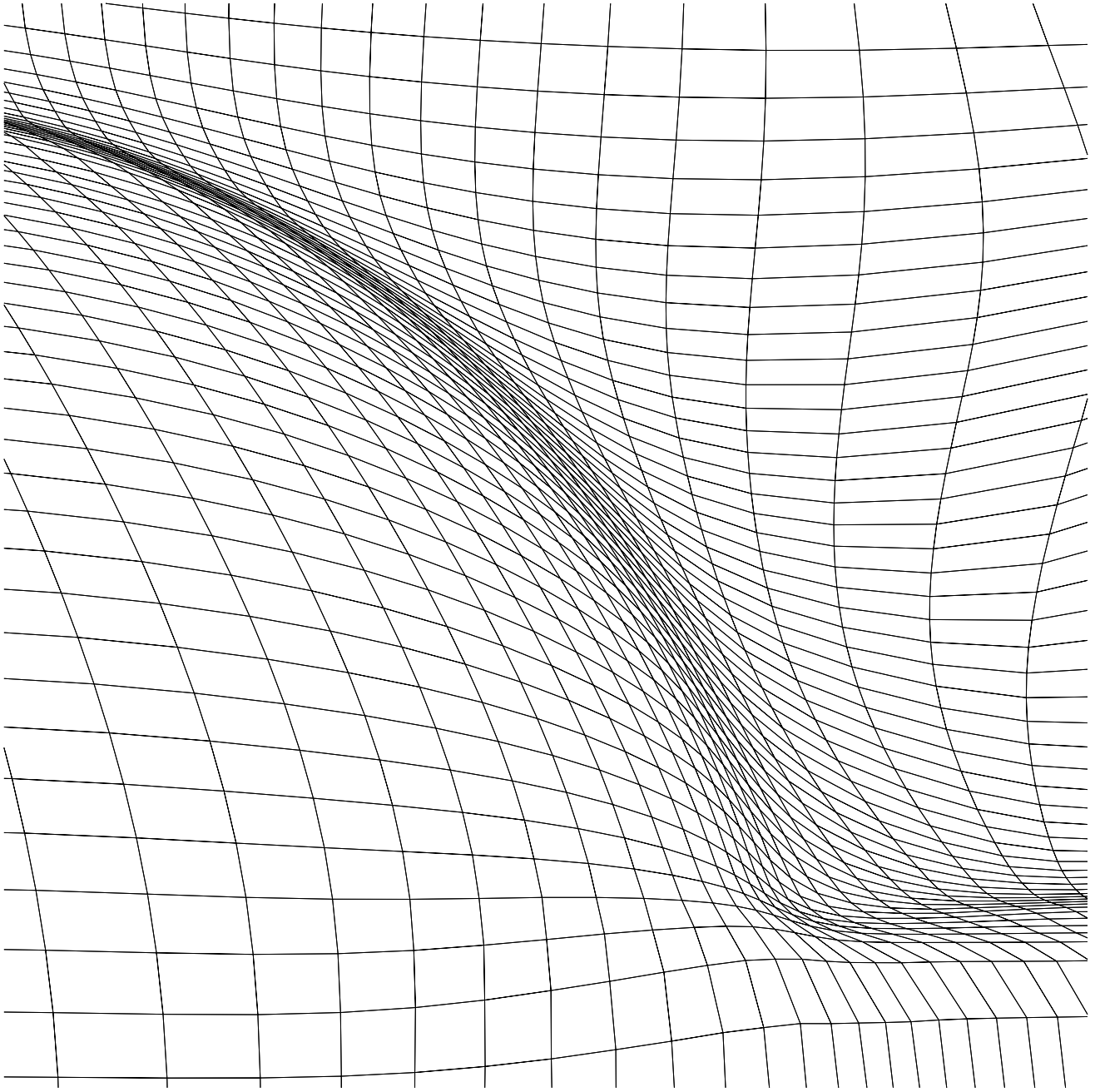
Patchwise ‘deformation’ adaptivity



Patchwise ‘deformation’ adaptivity



Patchwise ‘deformation’ adaptivity



Concepts for Iterative Solvers

1) MG: *too few arithmetic work vs. data exchange*

- parallelization of ‘recursive’ smoothers (only blockwise)?
- complicated geometric structures with local anisotropies?

2) DD: *bad convergence behaviour compared with MG*

- good ratio for communication/arithmetic work!
- implementation (overlap, coarse grid problem, **3D**)?

$$\Downarrow$$

1) + 2) = SCARC

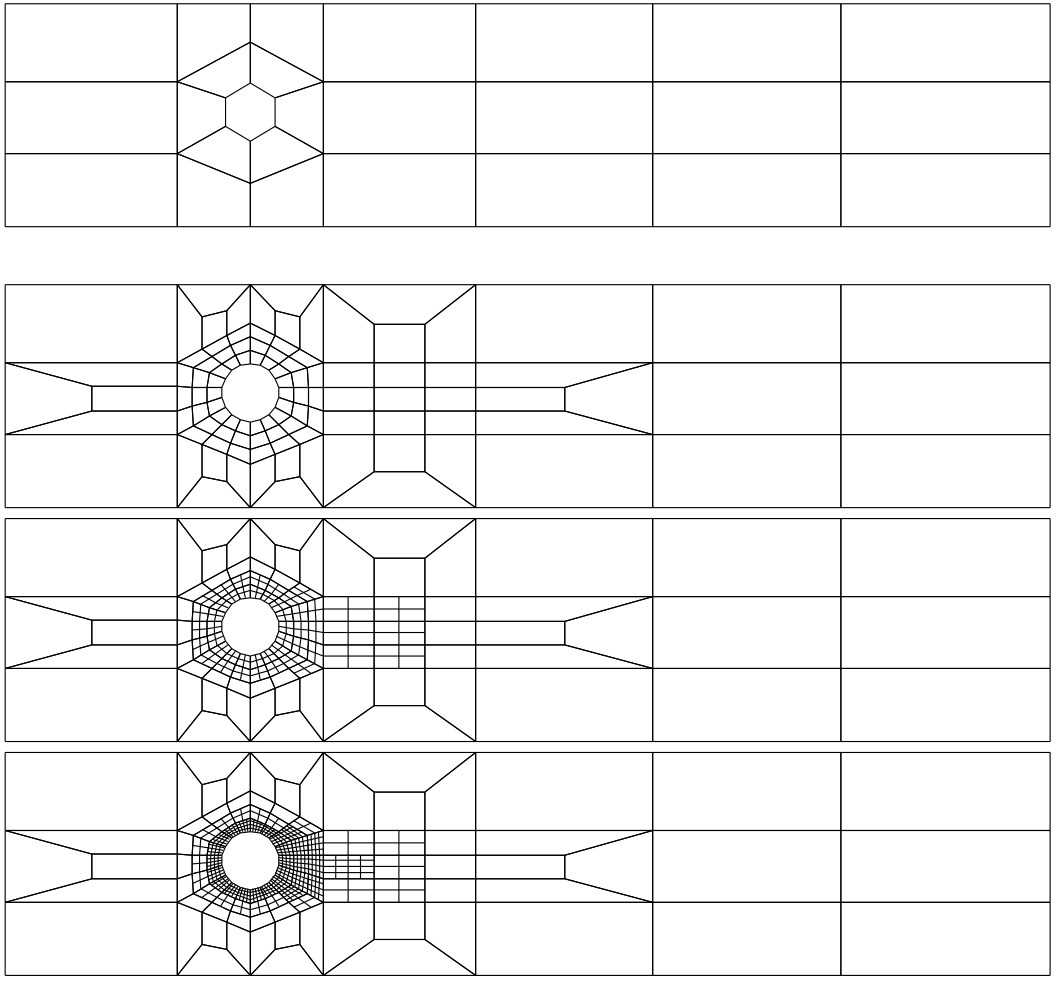
Hide recursively all anisotropies in ‘local’ units! (**robustness**)
Perform all Linear Algebra tasks on ‘local’ units only! (**efficiency**)

Summary for Adaptive ScaRC (I)

Parallel convergence rates for ScaRC-CG (2 global smoothing, V-cycle; 2 local 'MG-TriGS', F-cycle) with direct coarse mesh solver

MESH	#NEQ	AR	$\rho\,(\#IT)$
25-3	1,704	3	0.03 (5)
25-4	6,608	3	0.03 (4)
25-5	26,016	3	0.03 (4)
25-5-1	44,544	3	0.03 (5)
25-6-1	177,152	3	0.04 (5)
25-6-1-an	177,152	10^5	0.09 (6)
105-6	431,317	10	0.09 (6)
105-7	1,722,773	10	0.08 (6)
105-7-1	4,034,581	10	0.09 (6)
105-7-2	9,344,533	10	0.09 (6)
105-7-2-an	9,344,533	10^6	0.09 (6)
105-8-2-an	37,366,805	10^7	0.09 (6)

(Parallel) Global convergence



Mesh types

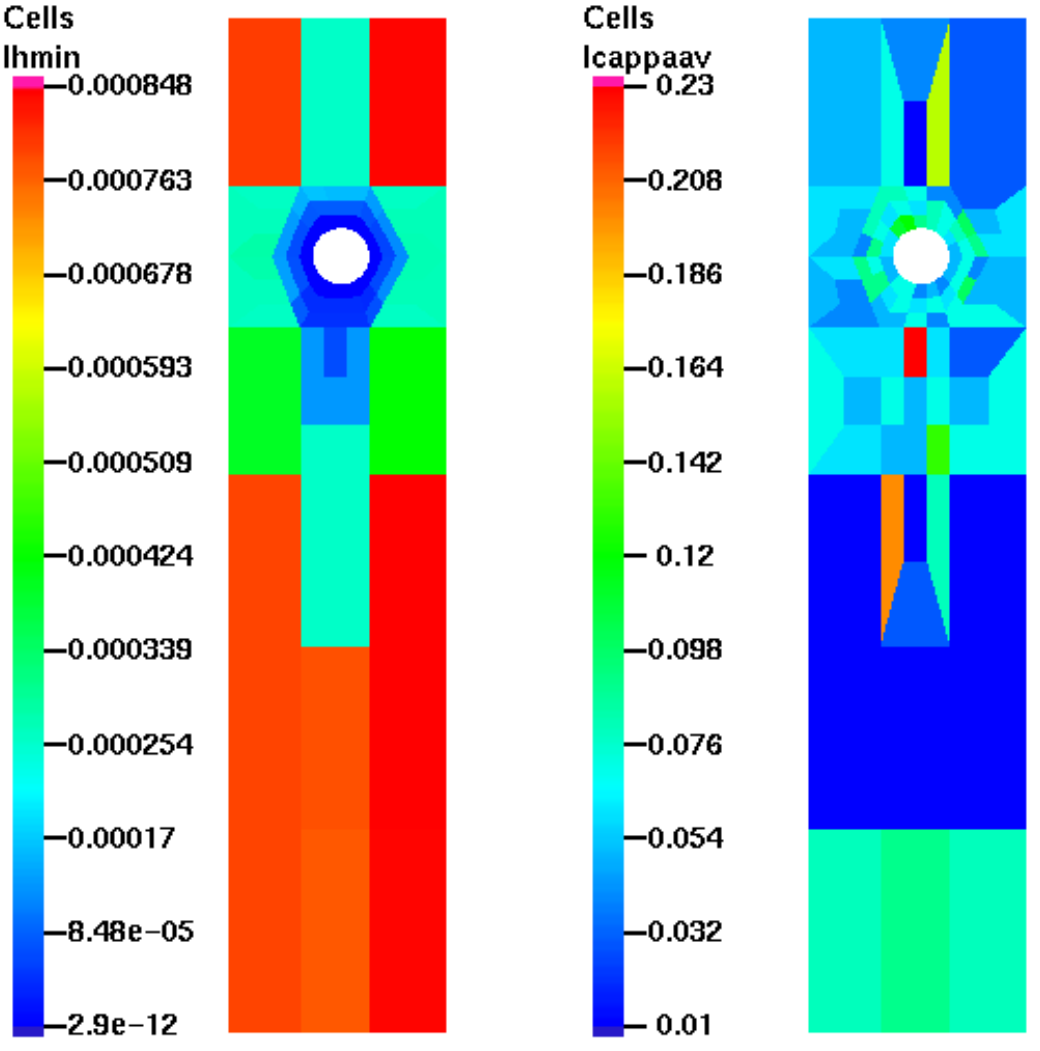
Summary for Adaptive SCARC (II)

#NVT	MV-V/C	MGTRI-V/C
65^2	788/1301	558/706
257^2	396/1255	434/672
1025^2	158/542	163/357

DEC 21264 (833 MHz) 'ES40'

#NVT	MV-V/C	MGTRI-V/C
65^2	258/1275	299/706
257^2	187/638	173/382
1025^2	179/630	146/288

AMD K7 (1333 MHz) 'XP+'



Local MFLOP/s rates

Local behaviour for 105-8-2-an

Open (Numerical) Problems

- **'Optimality' of the mesh ?**

  $\longrightarrow$ *w.r.t. number of unknowns or total CPU time ?*

- **Error estimators ?**

  $\longrightarrow$ *degree of macro-refinement ?* h/p*-refinement ?*

  $\longrightarrow$ *anisotropic deformation ? 'When to do what' decision ?*

- **Load balancing ?**

  $\longrightarrow$ *due to 'total CPU time per accuracy per processor' ?*

  $\longrightarrow$ *dynamical a posteriori process ?*

(Preliminary) Conclusion

- **Numerical efficiency ?**

  $\rightarrow$ *OK*

- **Parallel efficiency ?**

  $\rightarrow$ *(OK)*

- **Sequential efficiency ?**

  $\rightarrow$ *almost OK for CPU*

- **”Peak” efficiency ?**

  $\rightarrow$ *NO*

  $\rightarrow$ *GPU-based FEM preprocessors*

New: GPU Computing (D. Goeddeke)

- Techniques and data layouts
- Performance of basic numerical linear algebra components
- Efficiency vs. accuracy using 16bit floating point arithmetics
- Towards ”numerical GFLOP/s”
- Discussion

Hardware

All tests have been implemented in OpenGL + Cg on a Windows box.

Hardware

All tests have been implemented in OpenGL + Cg on a Windows box.

Four different systems have been evaluated:

- AMD Opteron (1800 MHz) as CPU reference with SBBLAS benchmark linked to GOTO BLAS
- NVIDIA 5950Ultra (NV30, 450 MHz, 4 vertex–, 8 fragment pipelines, 256bit memory interface, 425 MHz GDDR)
- NVIDIA 6800 (NV40, 350 MHz, 5 vertex–, 12 fragment pipelines, 256bit memory interface, 500 MHz GDDR2)
- NVIDIA 6800GT (NV40, 350 MHz, 6 vertex–, 16 fragment pipelines, 256bit memory interface, 500 MHz GDDR2)

Techniques

All GPU implementations are based on the following techniques. Visit our homepage for detailed tutorials and code examples.

- Render-to-texture and ”ping-ponging” between double-sided offscreen surfaces for fast iteration-type algorithms with data reuse.
- All calculations are performed in the fragment pipeline, the vertex pipeline is used to generate data which is uniformly interpolated by the rasterizer (e.g. array indices).
- Multitexturing and multipass partitioning for maximum efficiency.

Data layouts

Current NVIDIA GPUs support the following three major render target formats:

Data layouts

Current NVIDIA GPUs support the following three major render target formats:

- one 32 bit floating point value (LUMINANCE, s23e8 IEEE-like, memory imprint 32 bits), used to store a single vector
- four 32 bit floating point values (RGBA32, s23e8, memory imprint 128 bits), used to ”solve four systems simultaneously” (no dependencies between different channels, but different data in each channel).
- four 16 bit floating point values (RGBA16, s10e5, memory imprint 64 bits), again to ”solve four systems simultaneously”.

Remark: ATI only supports one to four 24 bit channels.

Numerical linear algebra (I)

The following low-level building blocks for FEM codes have been mapped to the GPU:

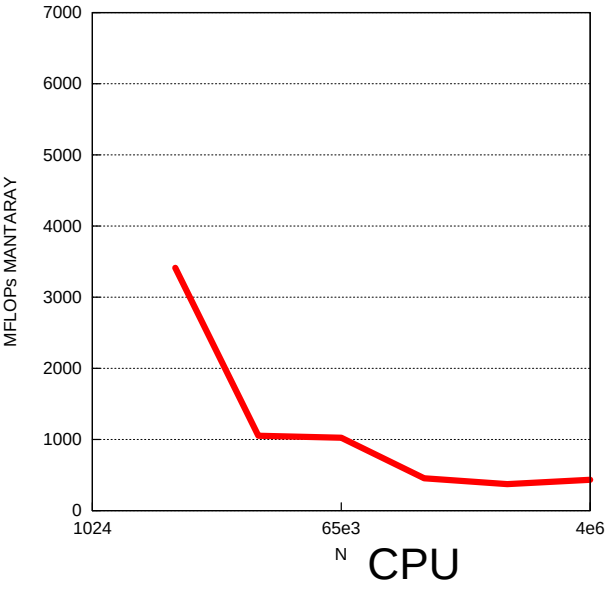
- BLAS SAXPY_C: $2N$ flops: $\mathbf{y}_i = \mathbf{y}_i + \alpha\mathbf{x}_i, \quad i = 1 \ldots N$
- SAXPY_V: $2N$ flops: $\mathbf{y}_i = \mathbf{y}_i + \mathbf{a}_i\mathbf{x}_i, \quad i = 1 \ldots N$
- MV_V for a 9-banded FEM (Q_1) matrix with variable coefficients: $18N$ flops, implemented as a series of 9 SAXPY_V operations with appropriate zero padding: $\mathbf{y} = \mathbf{y} + A\mathbf{x}$
- DOT: $2N$ flops, implemented as a logarithmic reduction: $y = \sum_{i=1}^{N} \mathbf{a}_i\mathbf{b}_i$
- NORM: $2N$ flops, implemented as a logarithmic reduction: $y = \sqrt{\sum_{i=1}^{N} \mathbf{a}_i\mathbf{a}_i}$

Numerical linear algebra (II)

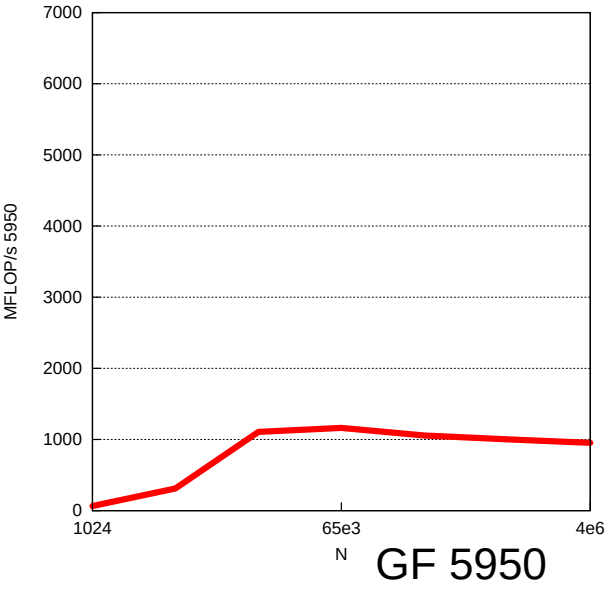
MFLOP/s rates for LUMINANCE data structure:

Numerical linear algebra (II)

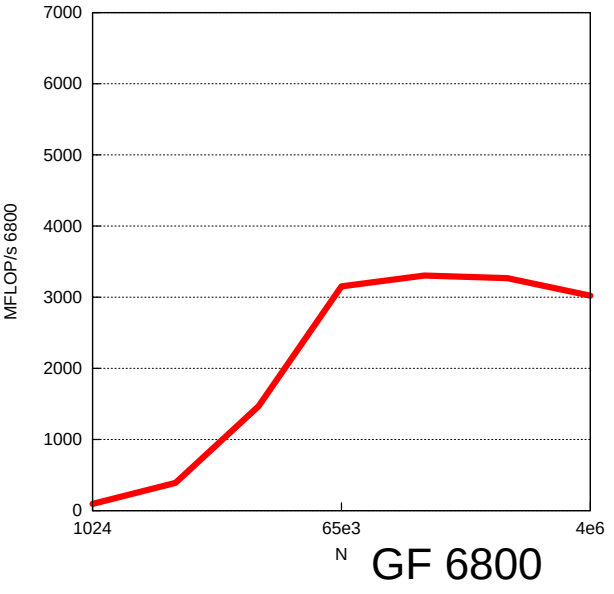
MFLOP/s rates for LUMINANCE data structure:



CPU



GF 5950

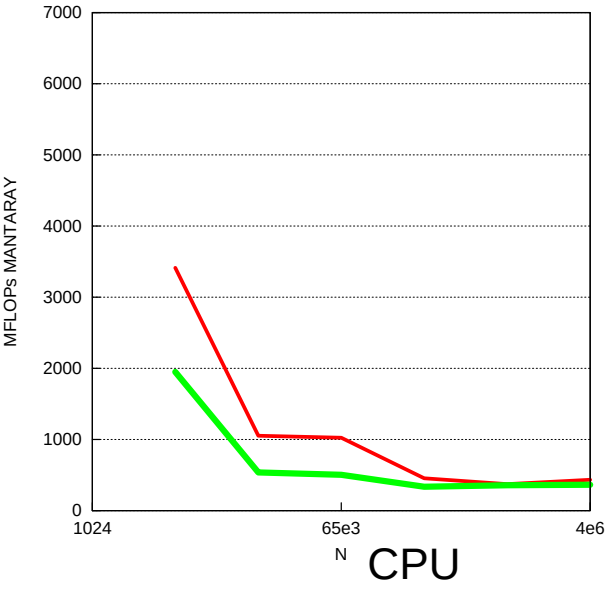


GF 6800

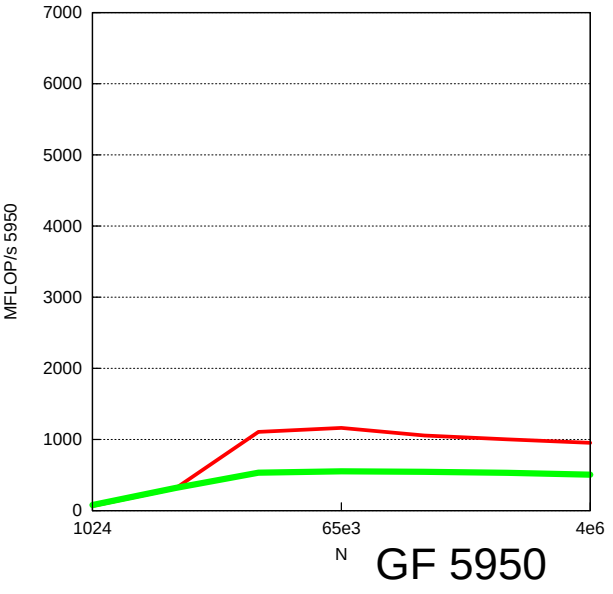
SAXPY_C

Numerical linear algebra (II)

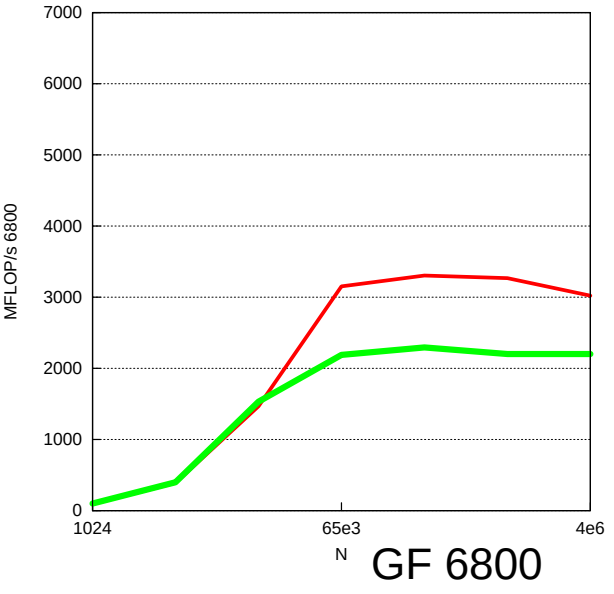
MFLOP/s rates for LUMINANCE data structure:



CPU



GF 5950

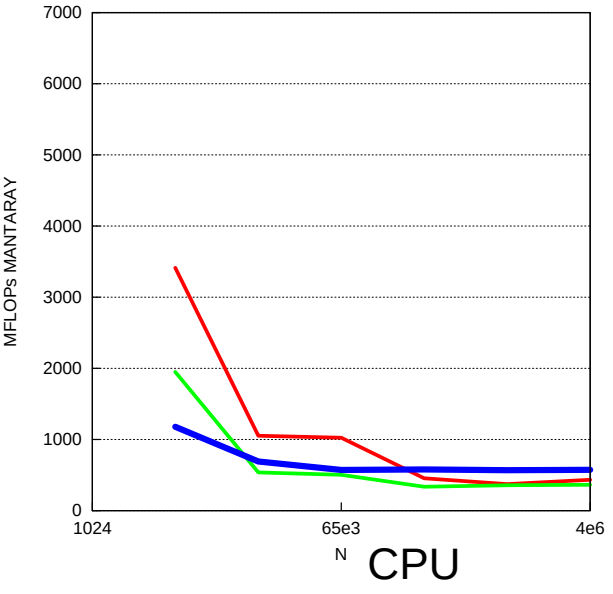


GF 6800

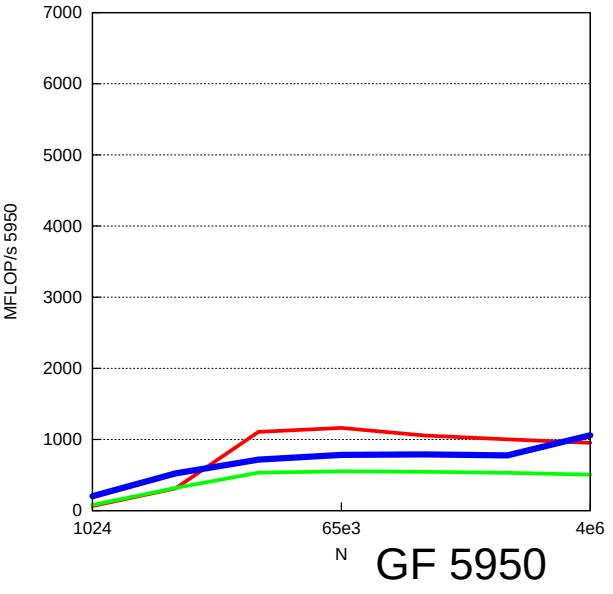
SAXPY_C SAXPY_V

Numerical linear algebra (II)

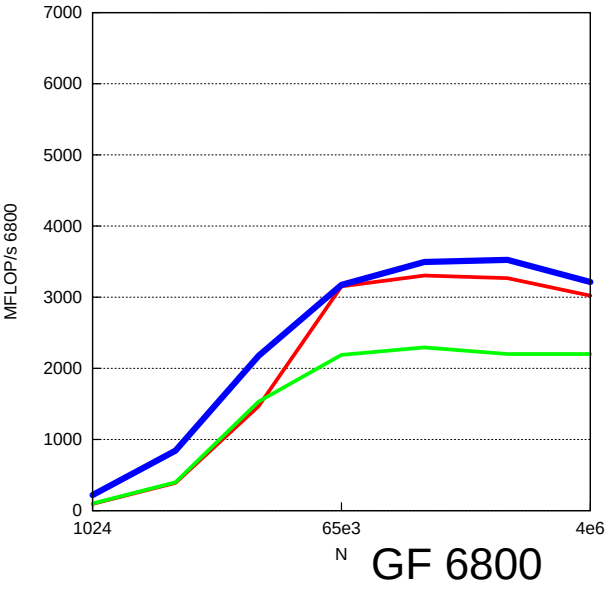
MFLOP/s rates for LUMINANCE data structure:



CPU



GF 5950

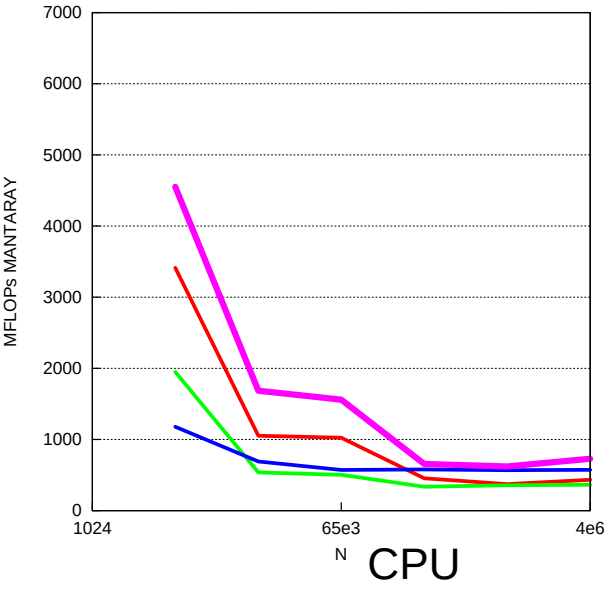


GF 6800

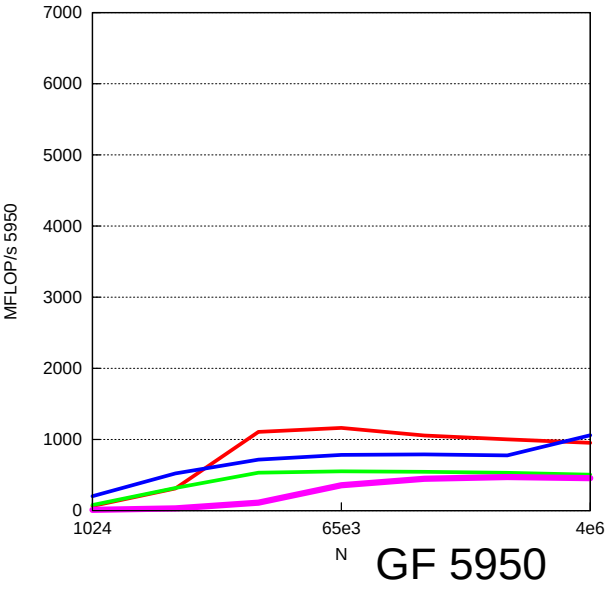
SAXPY_C SAXPY_V MV_V

Numerical linear algebra (II)

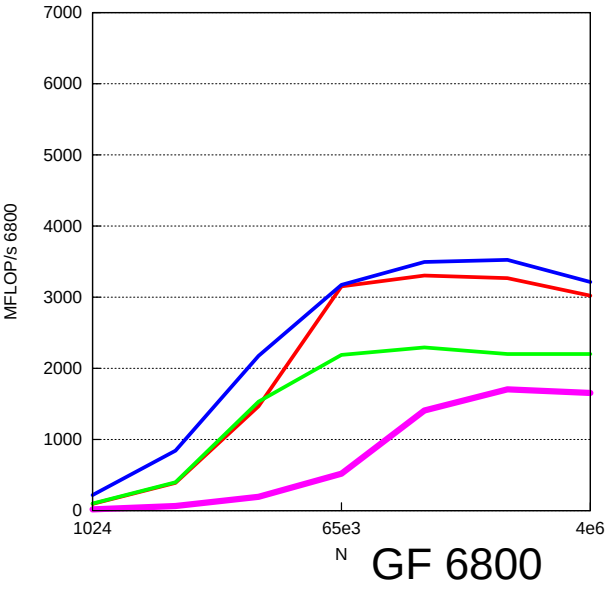
MFLOP/s rates for LUMINANCE data structure:



CPU



GF 5950

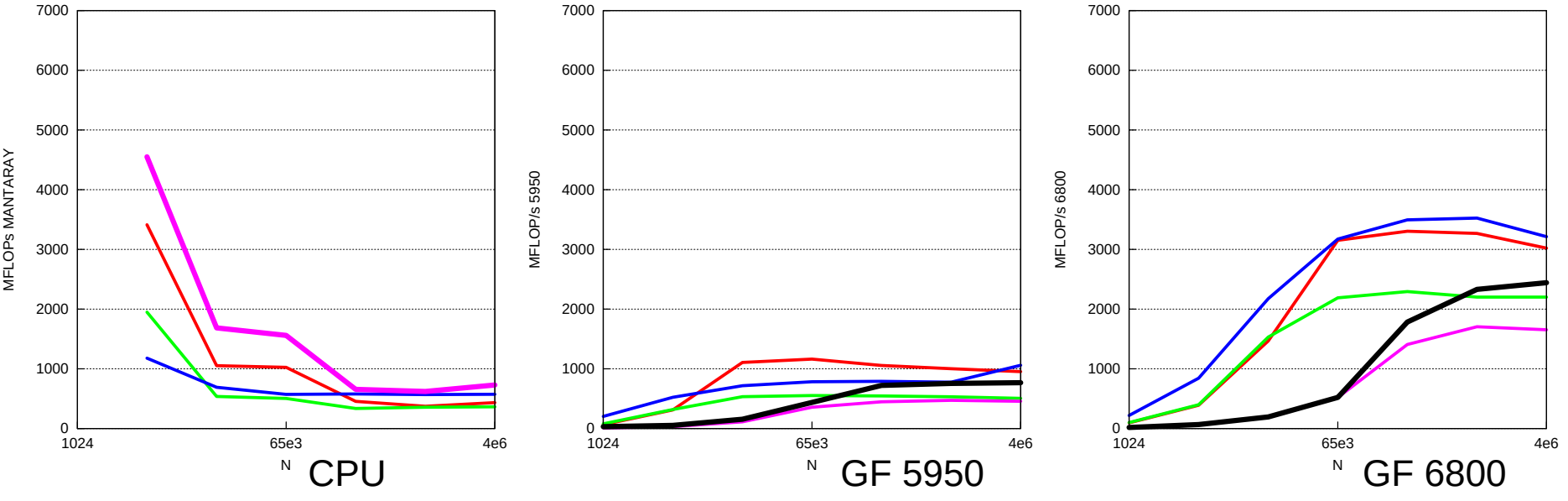


GF 6800

SAXPY_C SAXPY_V MV_V DOT

Numerical linear algebra (II)

MFLOP/s rates for LUMINANCE data structure:



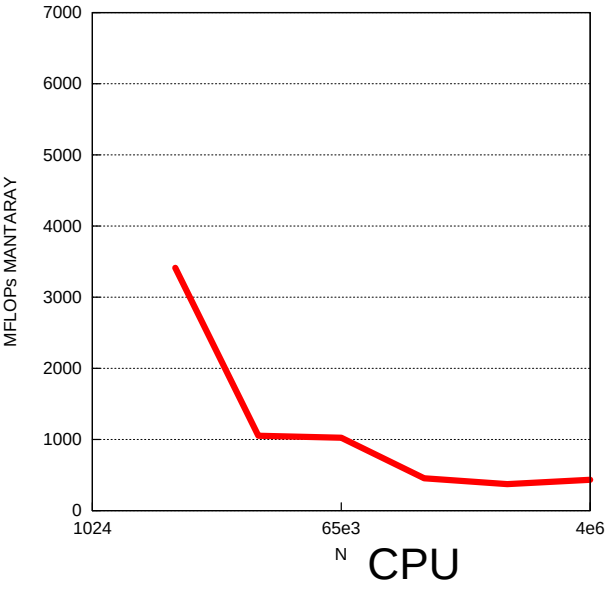
SAXPY_C SAXPY_V MV_V DOT NORM

Numerical linear algebra (III)

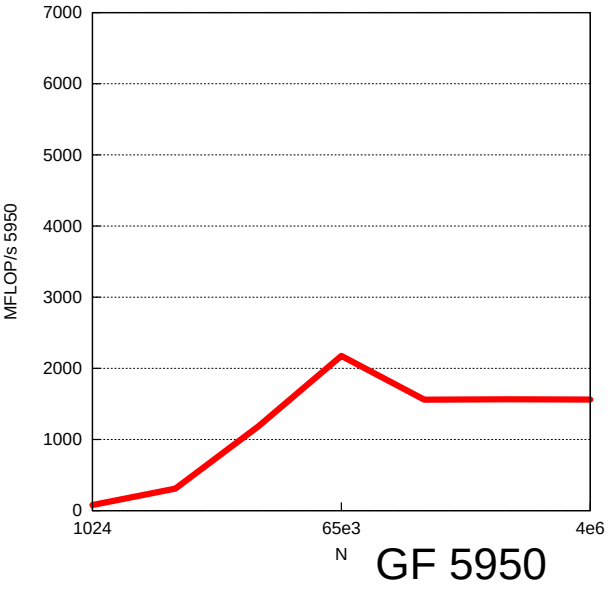
MFLOP/s rates for RGBA32 data structure:

Numerical linear algebra (III)

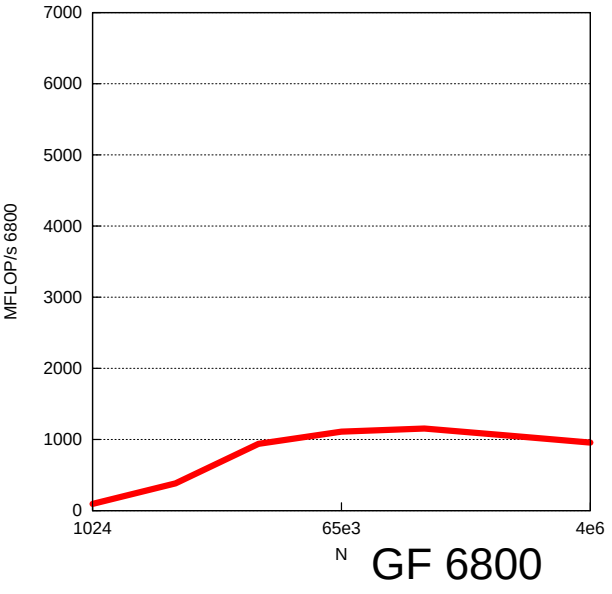
MFLOP/s rates for RGBA32 data structure:



CPU



GF 5950

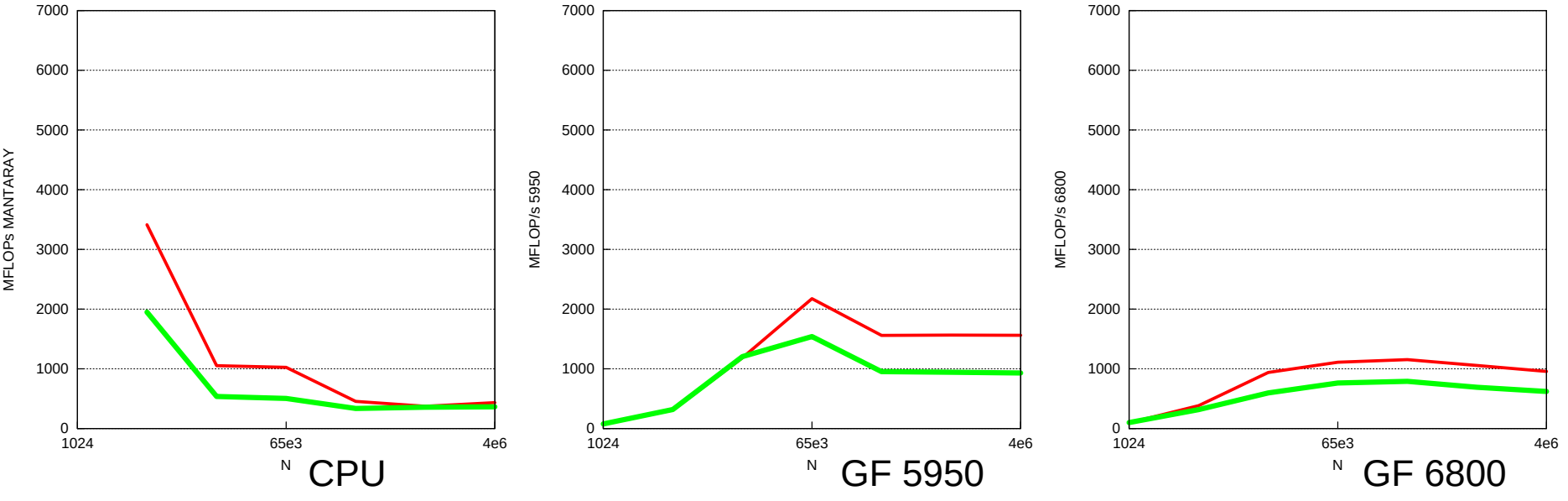


GF 6800

SAXPY_C

Numerical linear algebra (III)

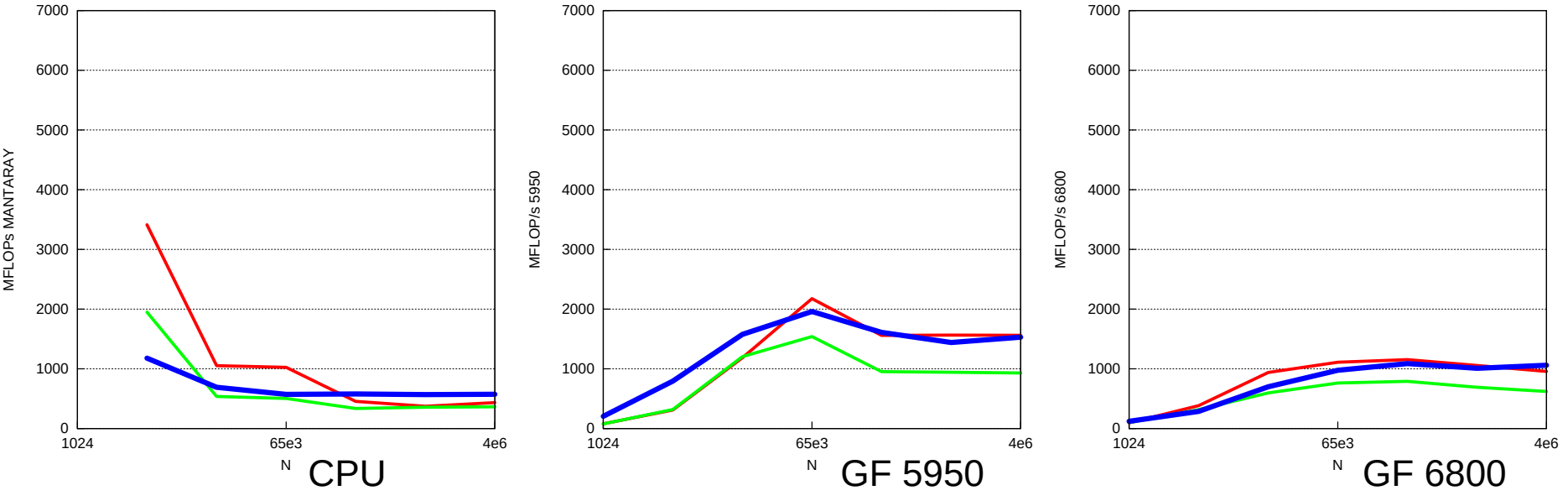
MFLOP/s rates for RGBA32 data structure:



SAXPY_C SAXPY_V

Numerical linear algebra (III)

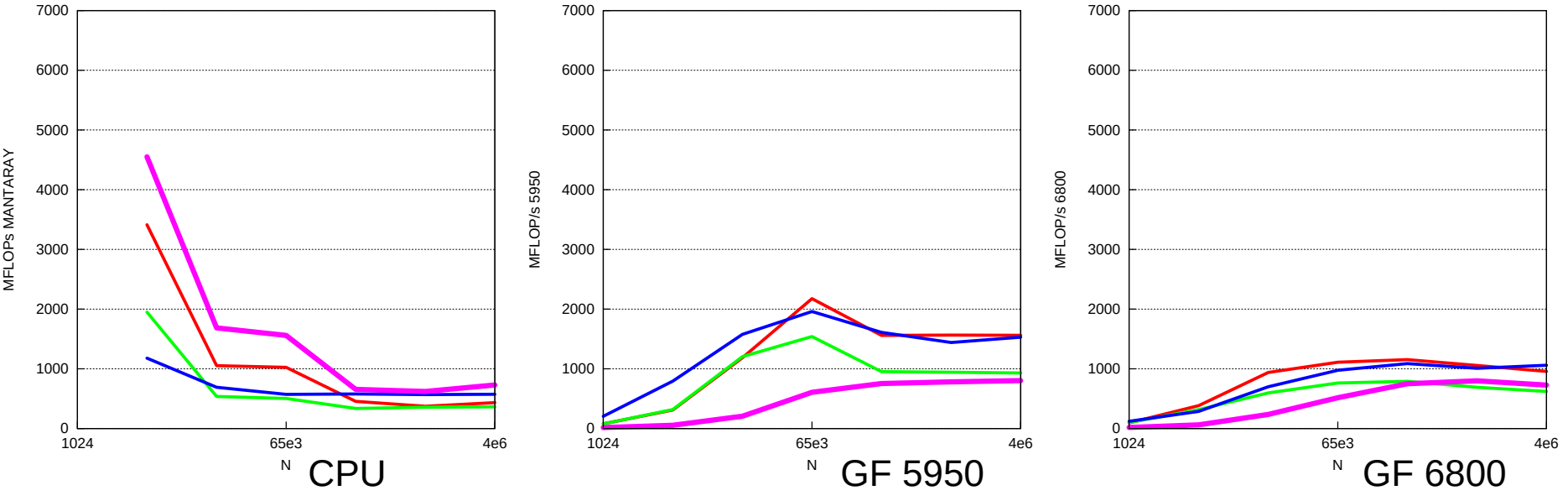
MFLOP/s rates for RGBA32 data structure:



SAXPY_C SAXPY_V MV_V

Numerical linear algebra (III)

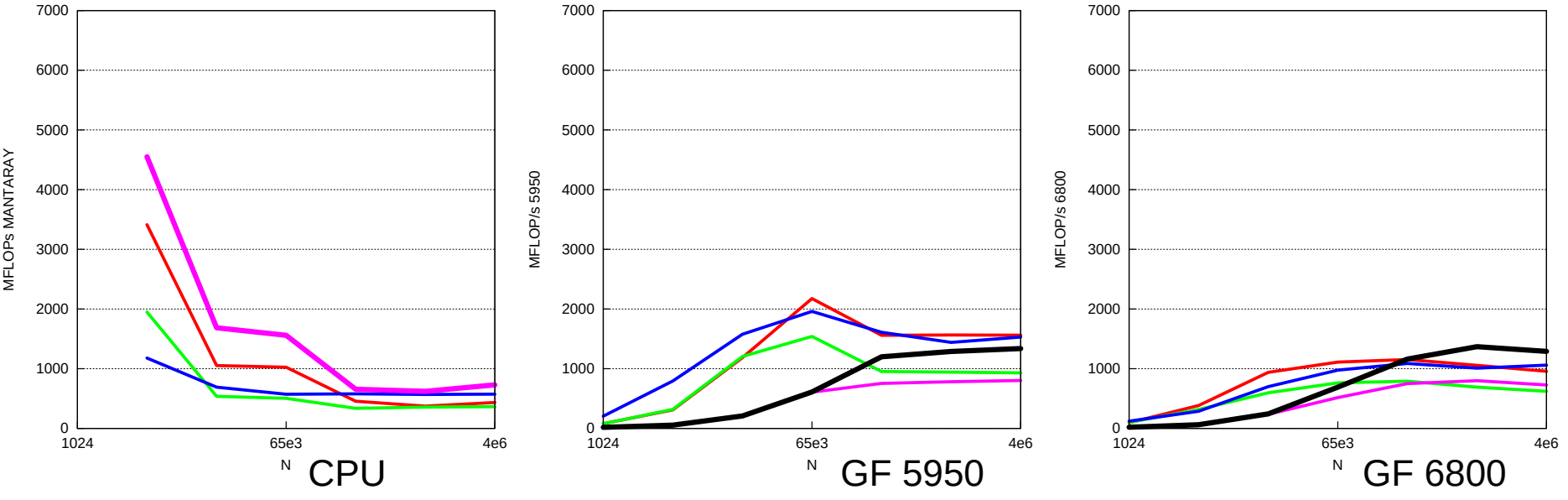
MFLOP/s rates for RGBA32 data structure:



SAXPY_C SAXPY_V MV_V DOT

Numerical linear algebra (III)

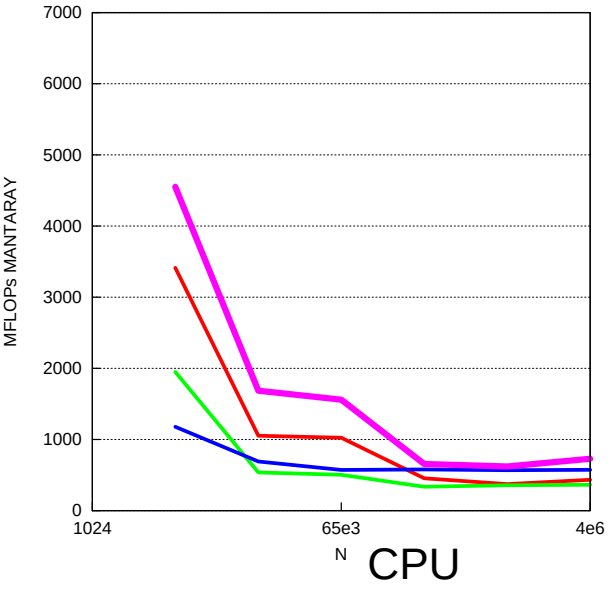
MFLOP/s rates for RGBA32 data structure:

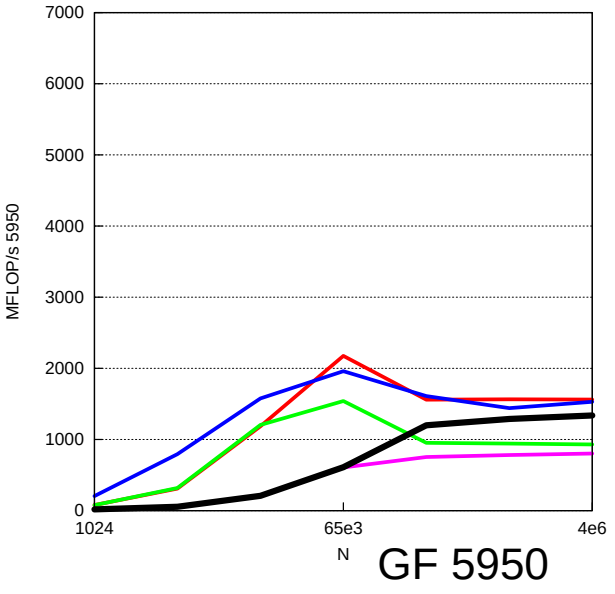


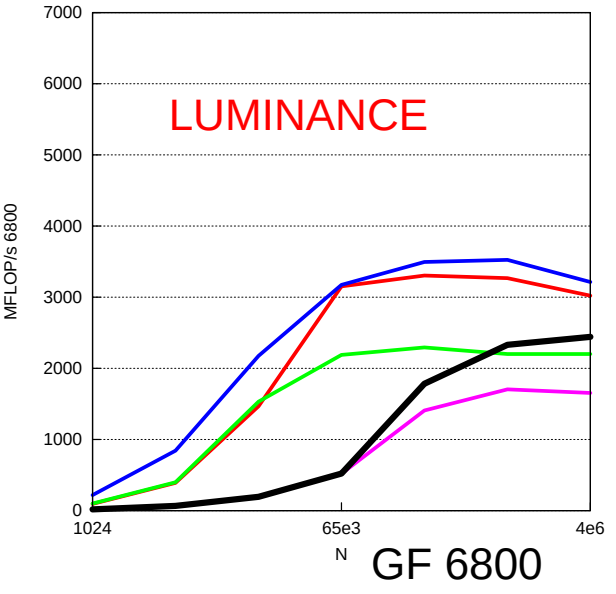
SAXPY_C SAXPY_V MV_V DOT NORM

Numerical linear algebra (III)

MFLOP/s rates for RGBA32 data structure:







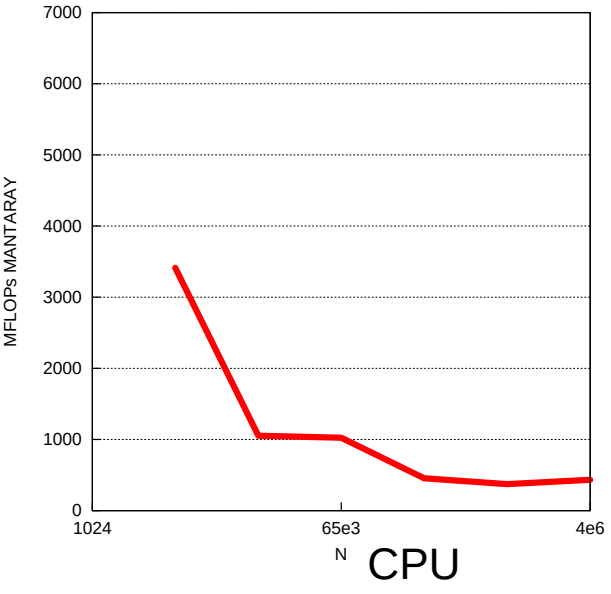
SAXPY_C SAXPY_V MV_V DOT NORM

Numerical linear algebra (IV)

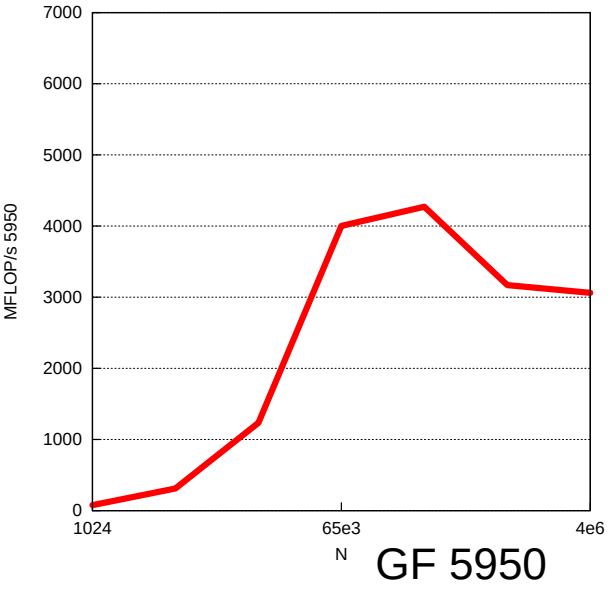
MFLOP/s rates for RGBA16 data structure:

Numerical linear algebra (IV)

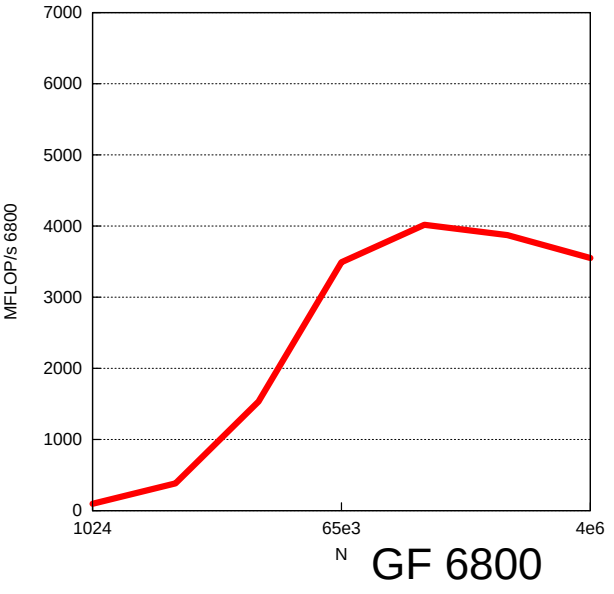
MFLOP/s rates for RGBA16 data structure:



CPU



GF 5950

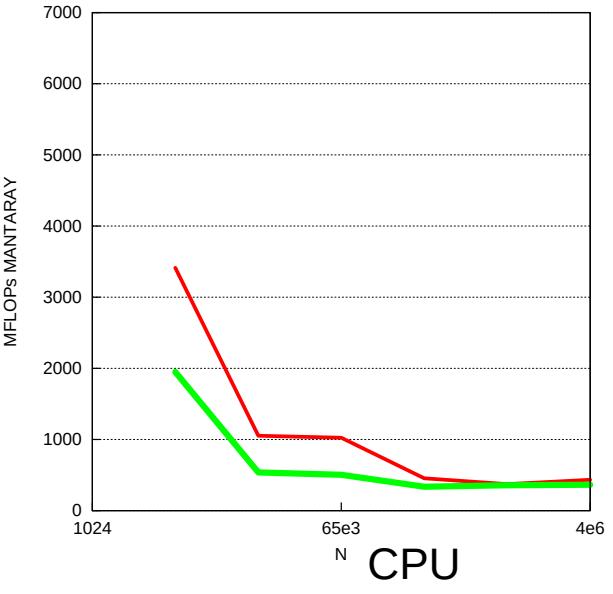


GF 6800

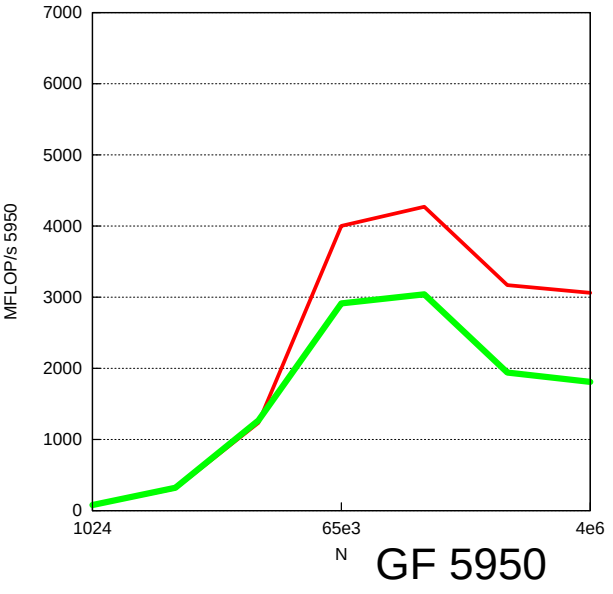
SAXPY_C

Numerical linear algebra (IV)

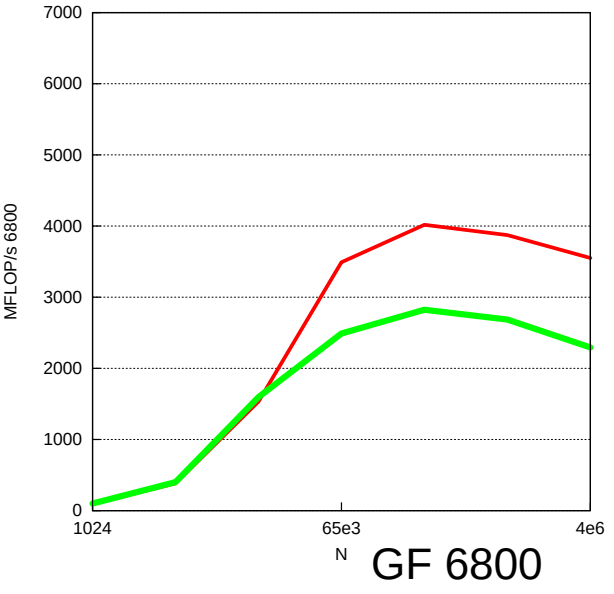
MFLOP/s rates for RGBA16 data structure:



CPU



GF 5950

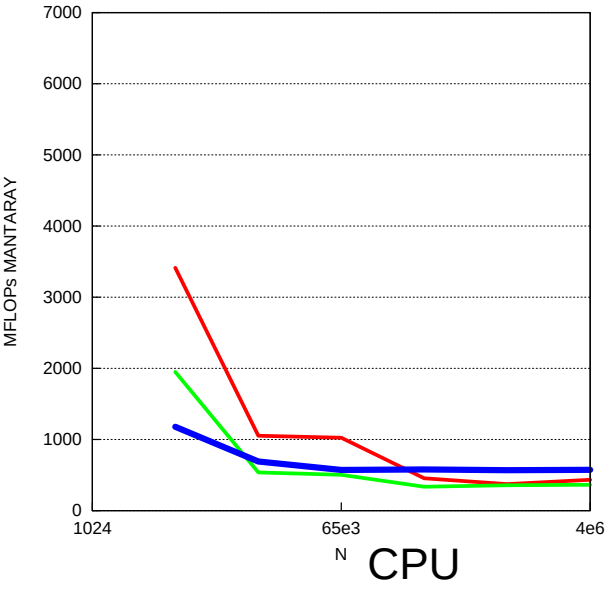


GF 6800

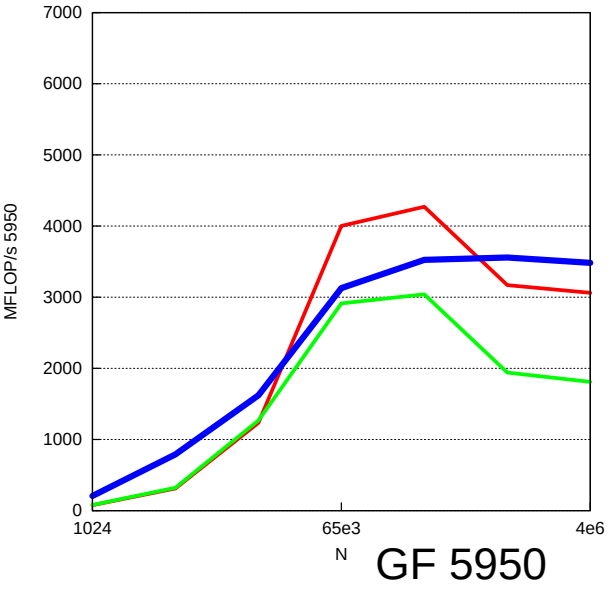
SAXPY_C SAXPY_V

Numerical linear algebra (IV)

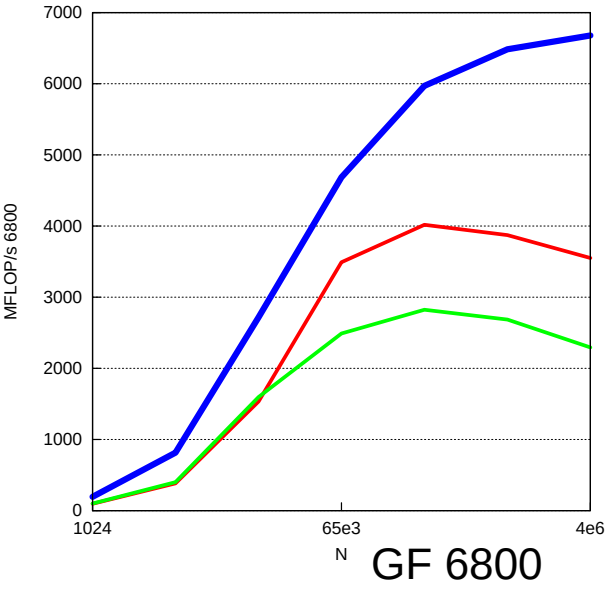
MFLOP/s rates for RGBA16 data structure:



CPU



GF 5950

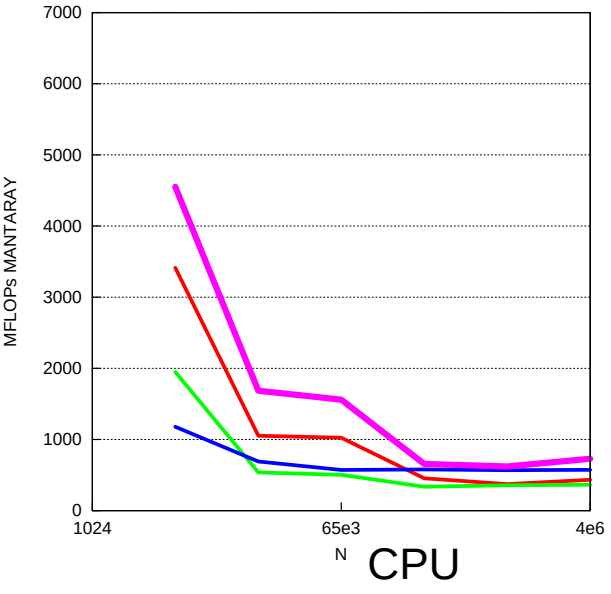


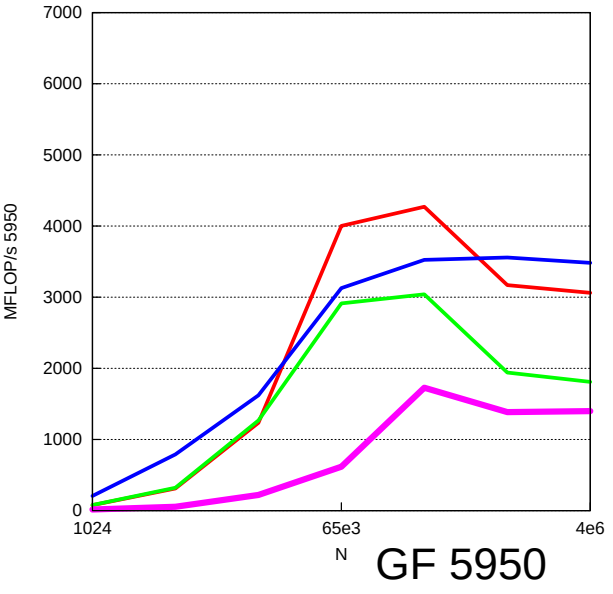
GF 6800

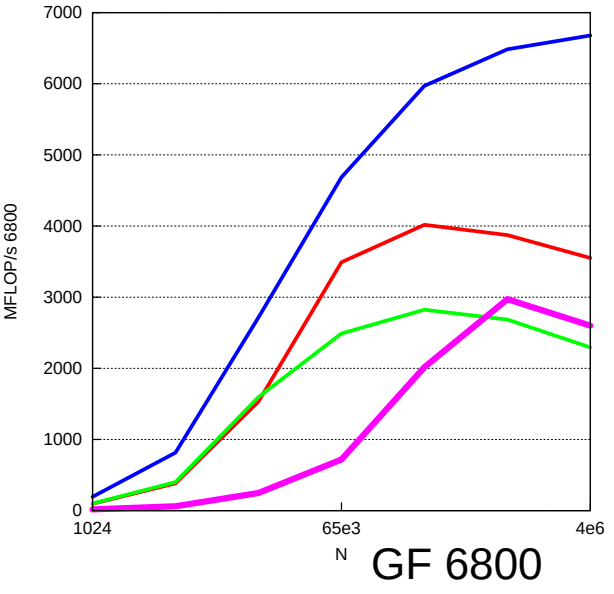
SAXPY_C SAXPY_V MV_V

Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:



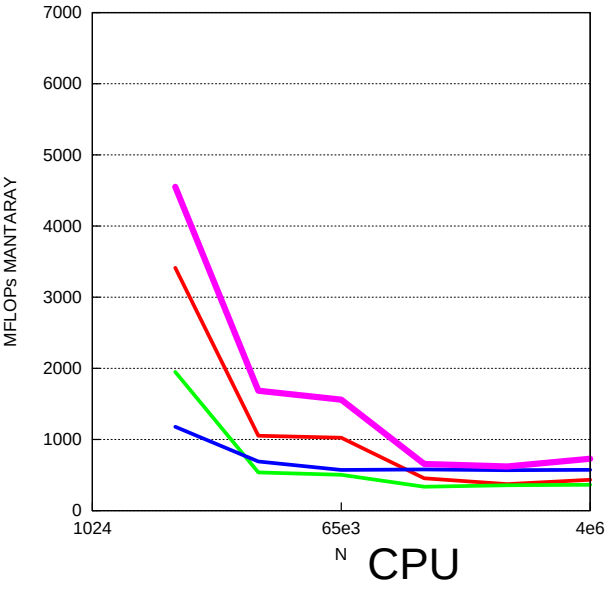


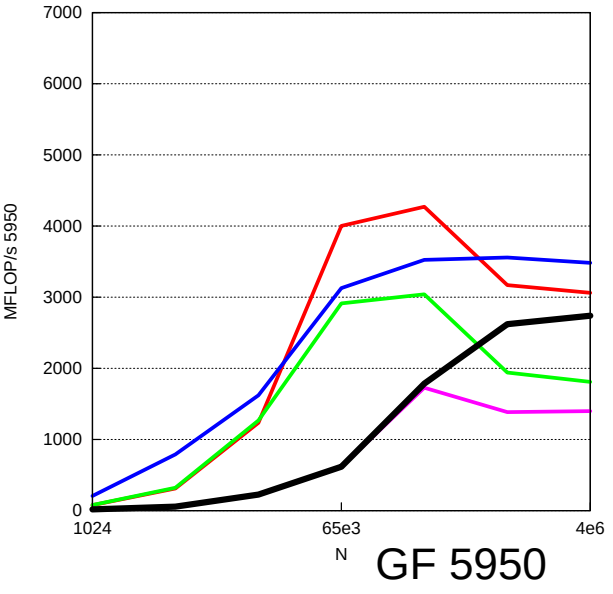


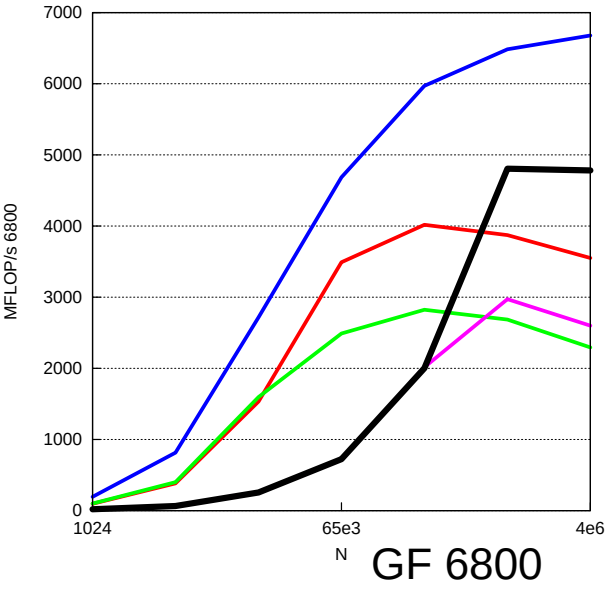
SAXPY_C SAXPY_V MV_V DOT

Numerical linear algebra (IV)

MFLOP/s rates for RGBA16 data structure:







SAXPY_C SAXPY_V MV_V DOT NORM

Conclusions and Questions

- GPUs outperform recent CPUs up to a factor of 5 for single precision arithmetics.
- GPUs only show their true potential for interesting problem sizes that crash the CPU cache.
- Different GPUs behave differently for solving single and quadruple tasks. Appropriate data layouts must be chosen independently for each GPU.
- GPU performance doubles to quadruples for 16 bit floating point arithmetics compared to 32 bit arithmetics.

Conclusions and Questions

- GPUs outperform recent CPUs up to a factor of 5 for single precision arithmetics.
- GPUs only show their true potential for interesting problem sizes that crash the CPU cache.
- Different GPUs behave differently for solving single and quadruple tasks. Appropriate data layouts must be chosen independently for each GPU.
- GPU performance doubles to quadruples for 16 bit floating point arithmetics compared to 32 bit arithmetics.

Questions:

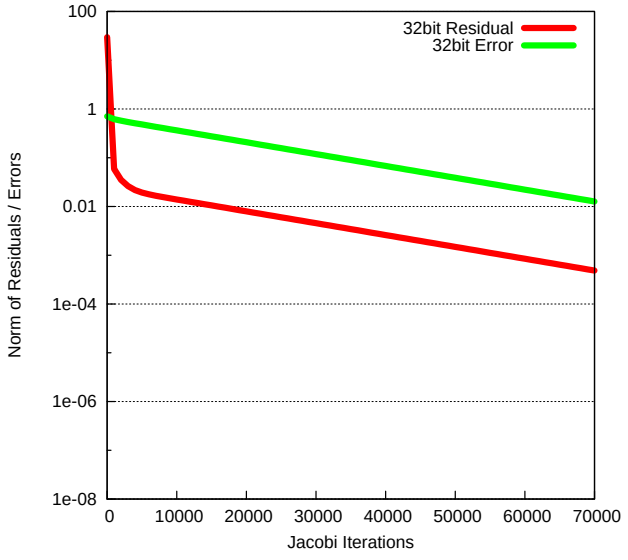
- How can the 16bit performance be achieved while maintaining 32bit accuracy?
- What about the 40 GFLOP/s that are announced?

Accuracy issues with 16 bits

Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA ("four independent systems simultaneously").

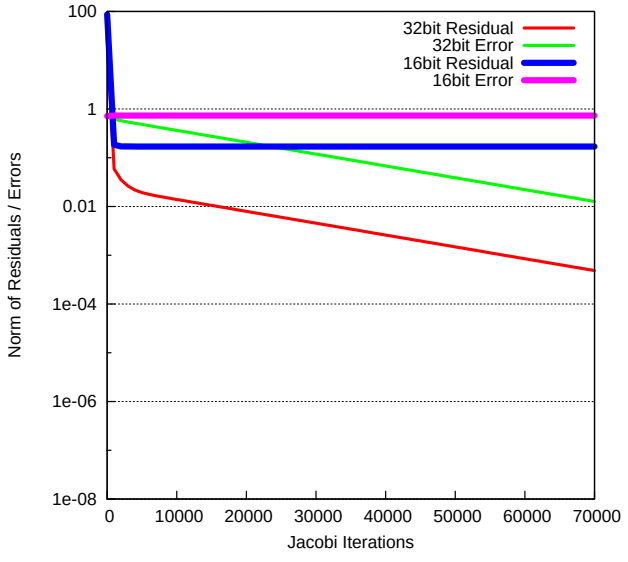
Accuracy issues with 16 bits

Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA ("four independent systems simultaneously").



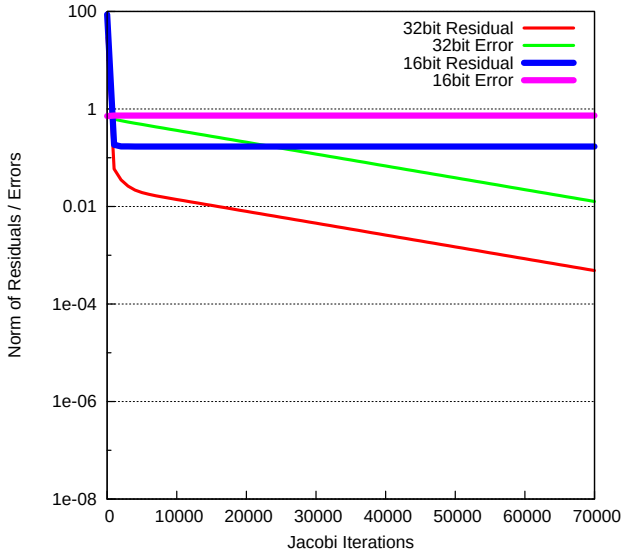
Accuracy issues with 16 bits

Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA ("four independent systems simultaneously").



Accuracy issues with 16 bits

Test case: Solve $A\mathbf{x} = \mathbf{0}$ with 9-band-stencil matrix A and random input $\mathbf{x}$. Use Jacobi scheme based on `MV_V` operator (as prototype for preconditioner and smoother in Krylov space methods) and RGBA ("four independent systems simultaneously").



"Half precision" floats are insufficient for applications beyond visual accuracy. But: Gaining at least one or two decimals is possible, making the use as preconditioner feasible!

Proof of concept

Use fast 16bit processing as preconditioner, update result ”occasionally” with single or double precision. All GPUs tested so far show identical floating point accuracy.

Proof of concept

Use fast 16bit processing as preconditioner, update result ”occasionally” with single or double precision. All GPUs tested so far show identical floating point accuracy.

1. Calculate defect: $\mathbf{d}^{(32)} = A^{(32)}\mathbf{x}^{(32)} - \mathbf{b}^{(32)},\ \alpha = ||\mathbf{d}^{(32)}||$.
2. Check some convergence criterion.

4. Shift solution: $\mathbf{b}^{(16)} = \mathbf{d}^{(32)}$, $\mathbf{x}^{(16)} = \mathbf{0}$.
5. Perform m Jacobi steps to ”solve” $A^{(16)}\mathbf{x}^{(16)} = \mathbf{b}^{(16)}$.
6. Shift corrected solution back: $\mathbf{x}^{(32)} = \mathbf{x}^{(32)} -$ $\mathbf{x}^{(16)}$.

Proof of concept

Use fast 16bit processing as preconditioner, update result ”occasionally” with single or double precision. All GPUs tested so far show identical floating point accuracy.

1. Calculate defect: $\mathbf{d}^{(32)} = A^{(32)}\mathbf{x}^{(32)} - \mathbf{b}^{(32)}$, $\alpha = ||\mathbf{d}^{(32)}||$. CPU or GPU
2. Check some convergence criterion.
   CPU

4. Shift solution: $\mathbf{b}^{(16)} = \mathbf{d}^{(32)}$, $\mathbf{x}^{(16)} = \mathbf{0}$.
   GPU or AGP transfer
5. Perform m Jacobi steps to ”solve” $A^{(16)}\mathbf{x}^{(16)} = \mathbf{b}^{(16)}$.
   GPU
6. Shift corrected solution back: $\mathbf{x}^{(32)} = \mathbf{x}^{(32)} - \quad \mathbf{x}^{(16)}$.
   GPU or AGP transfer

Proof of concept

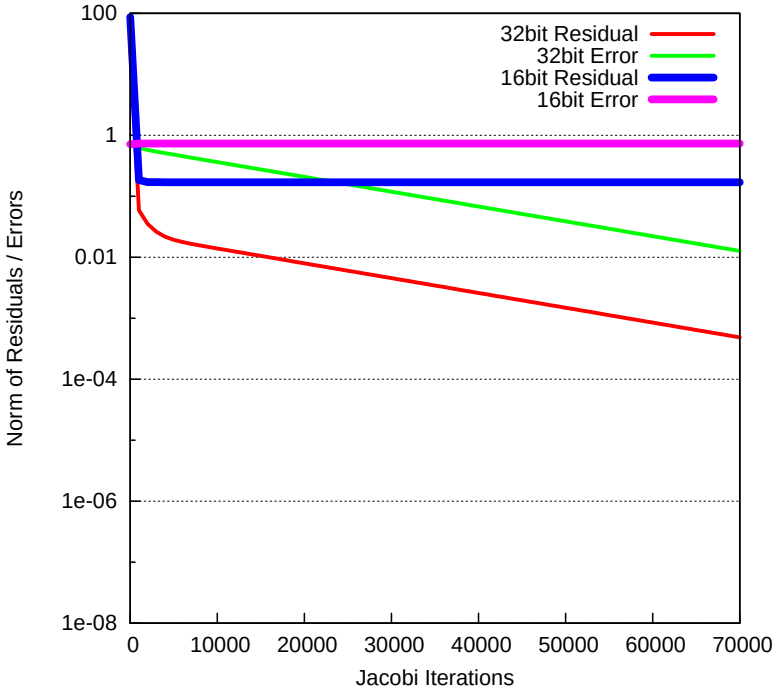
Use fast 16bit processing as preconditioner, update result ”occasionally” with single or double precision. All GPUs tested so far show identical floating point accuracy.

1. Calculate defect: $\mathbf{d}^{(32)} = A^{(32)}\mathbf{x}^{(32)} - \mathbf{b}^{(32)},\ \alpha = ||\mathbf{d}^{(32)}||$.
2. Check some convergence criterion.
3. Apply scaling by defect: $\mathbf{d}^{(32)} = 1/\alpha * \mathbf{d}^{(32)}$.
4. Shift solution: $\mathbf{b}^{(16)} = \mathbf{d}^{(32)}$, $\mathbf{x}^{(16)} = \mathbf{0}$.
5. Perform m Jacobi steps to ”solve” $A^{(16)}\mathbf{x}^{(16)} = \mathbf{b}^{(16)}$.
6. Shift corrected solution back: $\mathbf{x}^{(32)} = \mathbf{x}^{(32)} - \omega * \alpha * \mathbf{x}^{(16)}$.

Apply damping by ω and scaling by norm of defect for better convergence and to keep well within the dynamic range of the 16bit ”half precision” data type.

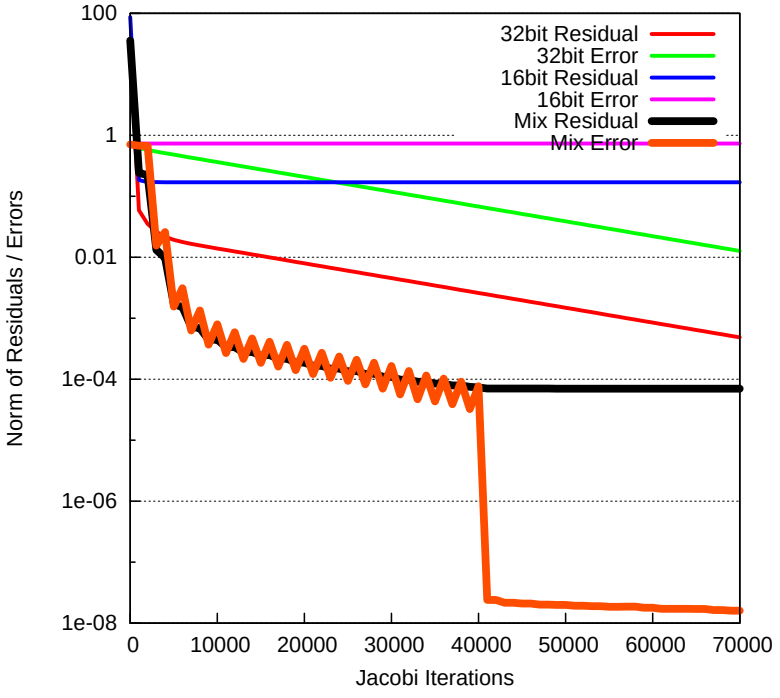
Proof of concept

Results:



Proof of concept

Results:



Proof of concept

Results:

- 32 bit Jacobi iteration: ~ 1100 MFLOPs, 70K iterations
- 16 bit Jacobi iteration: ~ 3800 MFLOPs, ∞ iterations
- Norm: ~ 2000 MFLOPs
- Combined scheme with correction on CPU: $\sim 800 - 1200$ MFLOPs (depending on problem size), 40K iterations
- Combined scheme running completely on GPU: ~ 3500 MFLOPs (independent of problem size), 40K iterations

Proof of concept

Results:

- 32 bit Jacobi iteration: ~ 1100 MFLOPs, 70K iterations
- 16 bit Jacobi iteration: ~ 3800 MFLOPs, ∞ iterations
- Norm: ~ 2000 MFLOPs
- Combined scheme with correction on CPU: $\sim 800 - 1200$ MFLOPs (depending on problem size), 40K iterations
- Combined scheme running completely on GPU: ~ 3500 MFLOPs (independent of problem size), 40K iterations

Questions:

- When should the update be performed?
- Can this be predicted a priori to avoid heavy data transfer to CPU?

How to get closer to peak performance?

”Data moving is expensive, not data processing” is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.

How to get closer to peak performance?

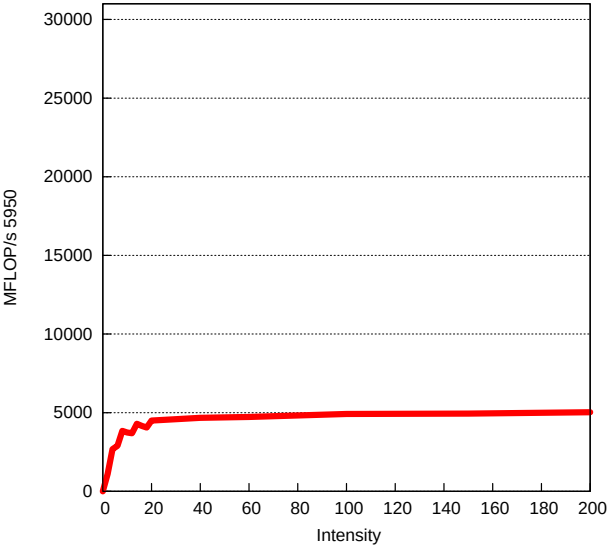
"Data moving is expensive, not data processing" is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.

Test case:

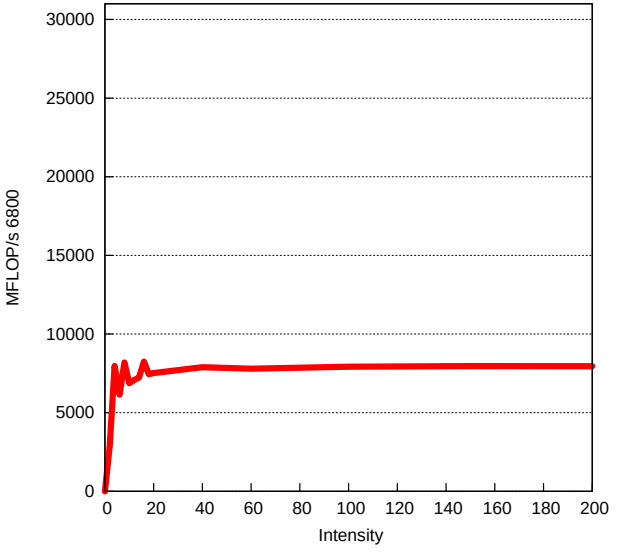
- Fetch value x_i from long vector / texture $\mathbf{x}$, $i = 1, \ldots, 1024^2$.
- Compute $x_i = x_i + x_i^2 + x_i^3 + x_i^4 + \ldots + x_i^m$.
- Rewrite Horner-style: degree m yields $2m - 1$ flops.

How to get closer to peak performance?

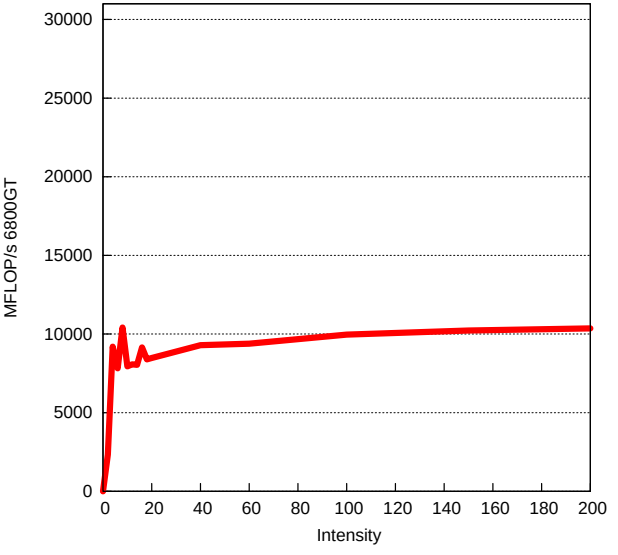
”Data moving is expensive, not data processing” is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.



GF 5950



GF 6800



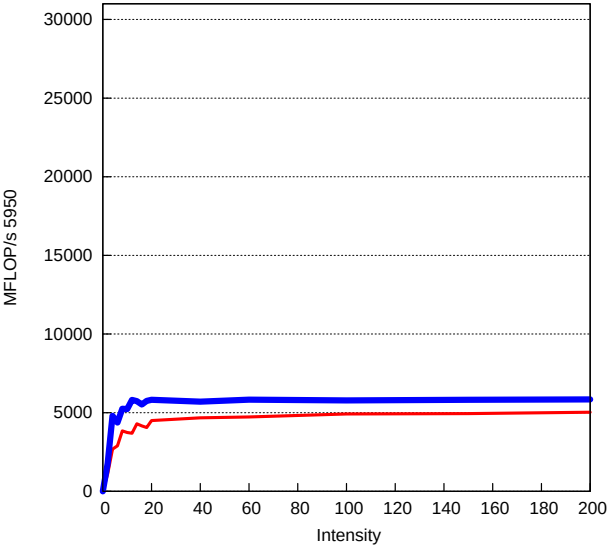
GF 6800GT*

LUMINANCE

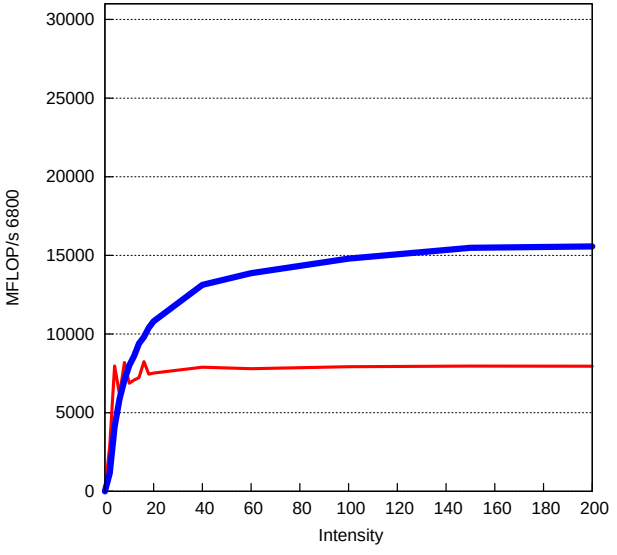
* courtesy of Hendrik Becker

How to get closer to peak performance?

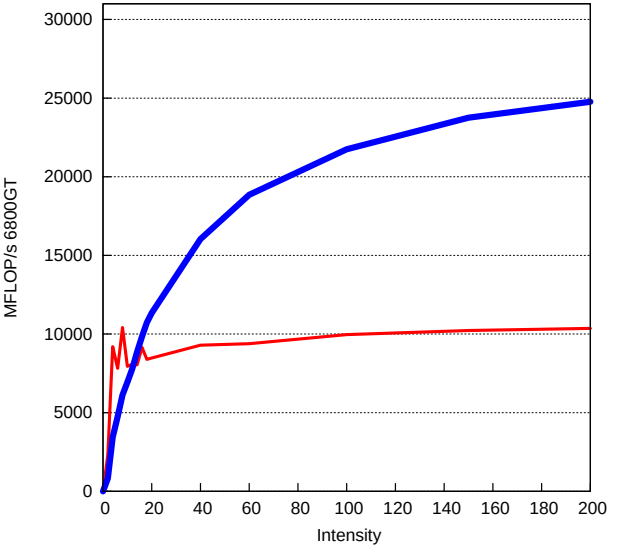
”Data moving is expensive, not data processing” is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.



GF 5950



GF 6800



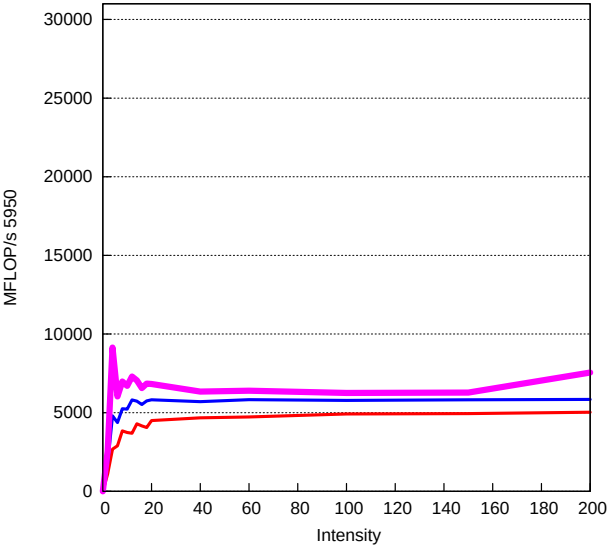
GF 6800GT*

LUMINANCE RGBA32

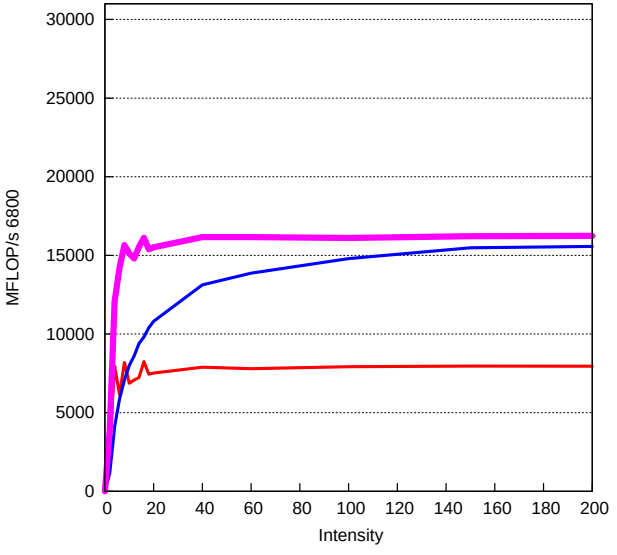
* courtesy of Hendrik Becker

How to get closer to peak performance?

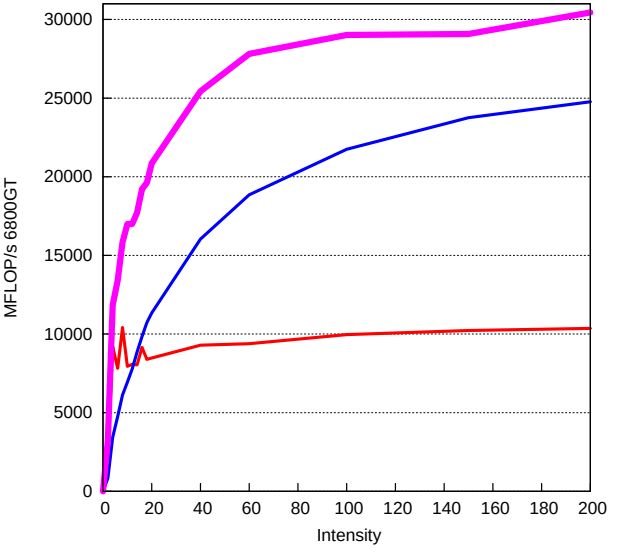
”Data moving is expensive, not data processing” is valid for GPUs as well! On GPUs, this can be quantified with the arithmetic intensity (number of flops per texture lookup) of implementations.



GF 5950



GF 6800



GF 6800GT*

LUMINANCE RGBA32 RGBA16

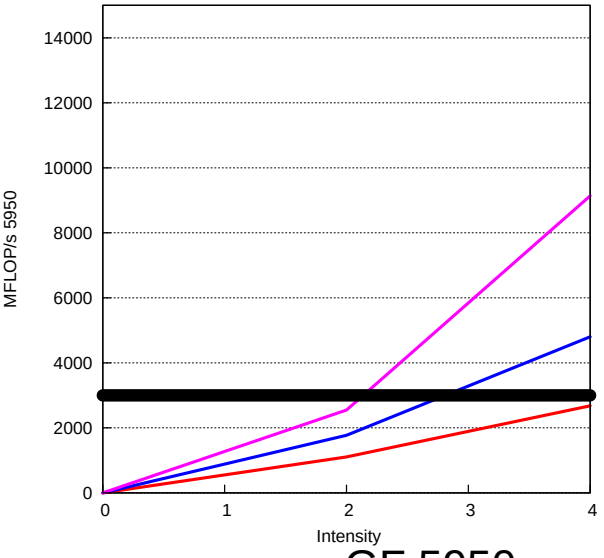
* courtesy of Hendrik Becker

How to get closer to peak performance?

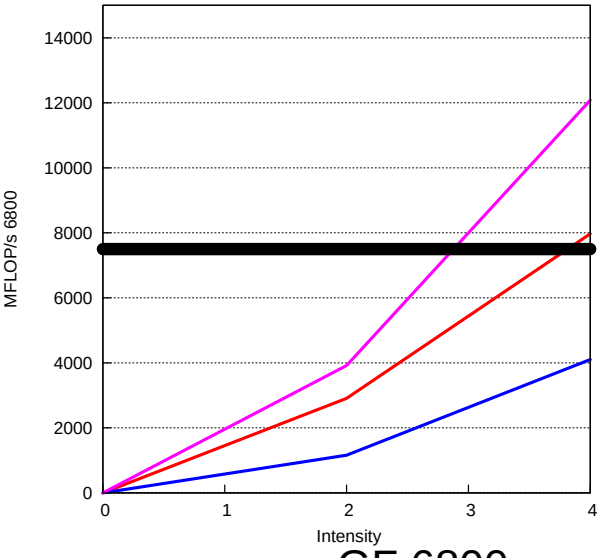
Realistic goal: 50% peak performance at a moderate intensity.

How to get closer to peak performance?

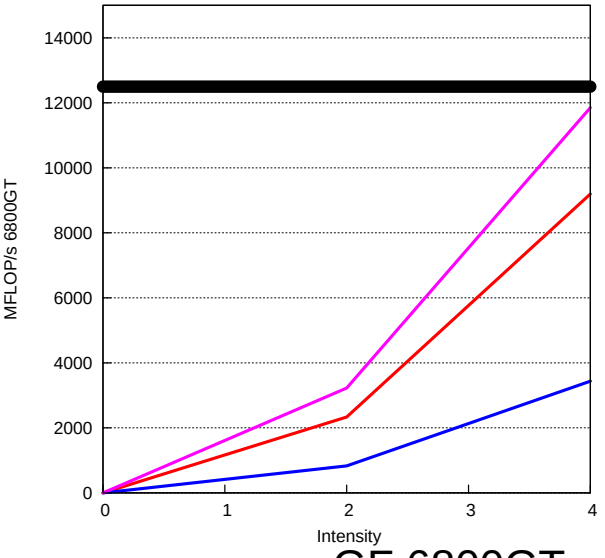
Realistic goal: 50% peak performance at a moderate intensity.



GF 5950



GF 6800

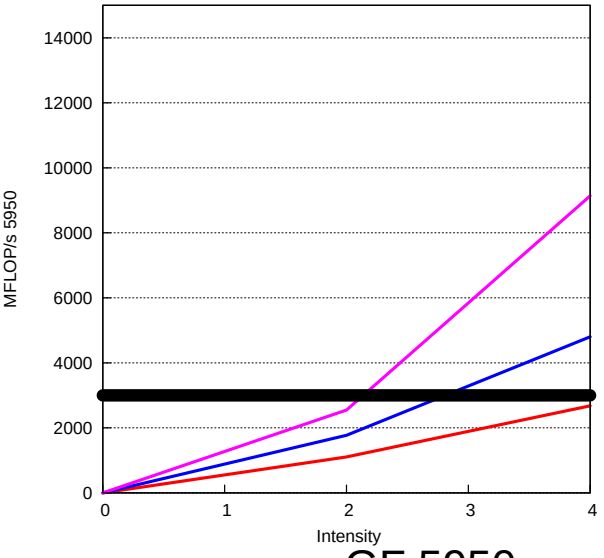


GF 6800GT

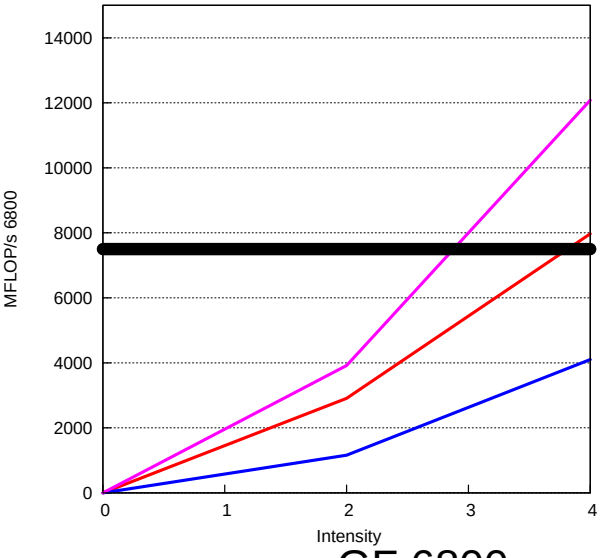
LUMINANCE RGBA32 RGBA16

How to get closer to peak performance?

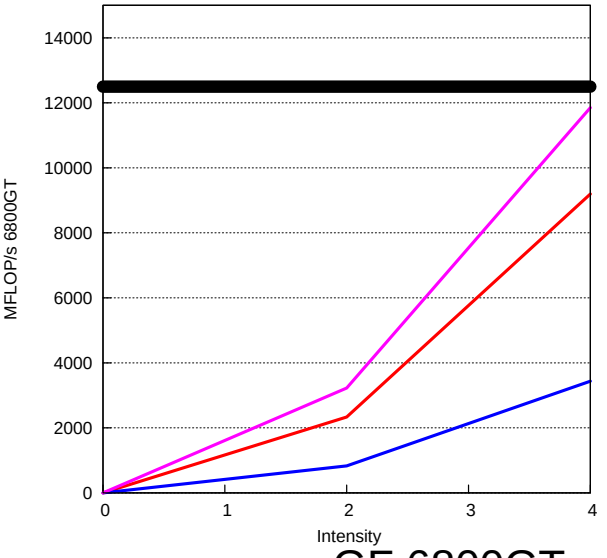
Realistic goal: 50% peak performance at a moderate intensity.



GF 5950



GF 6800



GF 6800GT

LUMINANCE RGBA32 RGBA16

Intensity of all examples presented so far is ≈ 1! GFLOP/s rate for `MV_V` and `JACOBI` is within 90% of the measured peak for this intensity.

Towards numerical GFLOP/s

Short term goal: Investigate further into using fast 16 bit preconditioning, this way working around the ”intensity barrier”.

Towards numerical GFLOP/s

Short term goal: Investigate further into using fast 16 bit preconditioning, this way working around the ”intensity barrier”.

Long term goal: Try to reformulate core FEM-multigrid components to increase their intensity: Assemble on-the-fly? Smart, complex preconditioners like ILU, ADI and SPAI?

Towards numerical GFLOP/s

Short term goal: Investigate further into using fast 16 bit preconditioning, this way working around the ”intensity barrier”.

Long term goal: Try to reformulate core FEM-multigrid components to increase their intensity: Assemble on-the-fly? Smart, complex preconditioners like ILU, ADI and SPAI?

Software goal: Don’t implement a full solver on the GPU, instead include the GPU as a fast preconditioner into the FEAST framework.

Conclusions

There is a huge potential for the future...

But: *Modern Numerics has to consider recent and future hardware trends!*

But: *Developing 'HPC-PDE software' is more than the implementation of existing Numerics for PDEs!*

Compromise between 'flexible/reusable/abstract' and 'machine-dependent/hardware-oriented' implementation style !

But: *Developing "production codes" requires lots of support and cooperation !*

Therefore: There is much to do...:-)